




中信出版社·CHINACITICPRESS

# 版权信息

书名:改变未来的九大算法

作者:[美]约翰·麦考密克

译者:管策

ISBN:9787508639017

中信出版集团制作发行

# 序

计算机行业正在改变我们的社会，正如物理学和化学在前两个世纪给社会带来的巨大改变一样。的确，数字技术几乎影响甚至颠覆了我们生活的方方面面。鉴于计算机行业对现代社会的重要性，人们对让这一切成为可能的基本概念却知之甚少，这显得有点自相矛盾。对这些概念的研究是计算机科学的核心，而这本麦考密克的新书则是向大众展示这些概念的少数书籍之一。

人们较少视计算机科学为一门学科，其中一个原因是，高中极少开设计算机科学这门课程。虽然人们通常认为要强制开设物理学和化学基础，但作为独立学科的计算机科学，通常只有在大学阶段才开设此课程。况且，学校讲授的“计算机”或“信息与通信技术”知识，通常只是略高于使用软件的技能训练。因此，学生们认为计算机学科枯燥也并不意外；而他们在娱乐和通信上使用计算机技术的天然热情，也被创造这类技术缺乏学术深度的印象所减弱。这些问题被认为是导致过去10年大学计算机科学专业学生人数下降一半的核心原因。考虑到数字技术对现代社会的极度重要性，让人们重新领略计算机科学的奇妙之处已经刻不容缓。

2008年，我很荣幸地被选为第180届英国皇家学院圣诞讲座（Royal Institution Christmas Lectures）的演讲人，该讲座由迈克尔·法拉第（Michael Faraday）于1826年发起。2008年圣诞讲座的主题首次涉及计算机科学。在准备这些讲座时，我花了很多时间来思考如何向大众解释计算机科学，却发现满足这一需求的资源很少，几乎没有关于计算机科学的畅销书。因此，我特别高兴能看到麦考密克的这本新书。

麦考密克在面向大众介绍计算机科学的复杂思想上做得非常好。这其中许多思想极其新颖，仅从这点上来看，它们就很值得关注。举个例子：电子商务的爆炸式增长之所以成为可能，是因为具备了能在互联网上秘密、安全地发送机密信息（如信用卡卡号）的能力。数十年来，建立在“开放”通道上的保密通信被认为是一个科学难题。当人们发现解决方法时，才发觉保密通信极度优雅，而麦考密克也以精确的类比进行了解释，无须读者拥有计算机科学知识。这些优点使这本书对科普读物做出了不可估量的贡献，我极力推荐本书。

克里斯·毕晓普（Chris Bishop）

微软剑桥研究院资深科学家

大不列颠皇家学院副院长

爱丁堡大学计算机科学教授

# 第一章　前言——计算机日常运用的卓越思想有哪些

此乃小技……为诗之诀在于有气、有势、有情、有韵、有起、有承、有转、有合。

——威廉姆·莎士比亚，
《爱的徒劳》（Love’s Labour’s Lost）

计算机科学中的伟大思想是如何诞生的？以下遴选部分思想进行介绍：

• 20世纪30年代，在第一台数字计算机发明以前，一位英国天才开创了计算机科学领域。之后，这位天才继续证明，不管未来建造的计算机运行多快、功能多强大、设计得多好，仍旧有一些问题是计算机不能解决的。

• 1948年，一位供职于电话公司的科学家发表了一篇论文，开创了信息理论领域。这位科学家的工作让计算机能以完美的精确度传输信息，即便大部分数据都因为被干扰而破坏。

• 1956年，一群学者在达特茅斯举行会议。这次会议的目标很清晰，也很大胆，那就是开创人工智能领域。在取得了许多重大成功以及经历了无数失望之后，我们仍期待出现一个真正的智能计算机程序。

• 1969年，IBM公司的一名研究人员发明了一种将信息组织进数据库中的先进方法。目前，绝大多数在线交易都使用该技术存储及检索信息。

• 1974年，英国政府秘密通信实验室的研究人员发明了一种让计算机安全通信的方法，即另一台计算机可以查看在计算机之间交换的所有信息。这些研究人员为政府保密所限——不过幸运的是，三名美国专家独立开发并拓展了这项重大发明，为互联网上所有的安全通信打下了基础。

• 1996年，两名斯坦福大学博士生决定联手搭建一个互联网搜索引擎。几年后，他们创办了谷歌公司——互联网时代的第一个数字巨头。

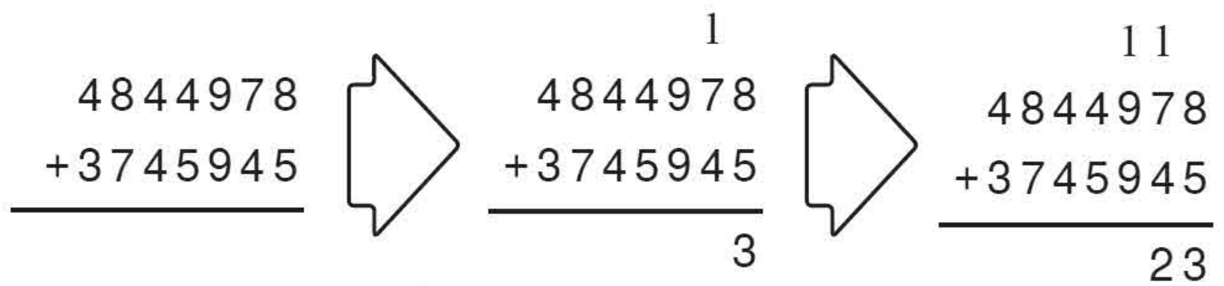
在我们享受21世纪技术惊人增长的同时，使用计算机设备——不管是现有最强大的一组机器还是最新、最时尚的手持设备——都不可避免地要依赖计算机科学的基础思想，而这些思想都诞生于20世纪。想一想：你今天做过什么令人印象深刻的事情吗？好吧，这个问题的答案取决于你怎么看。也许你搜索了包含数十亿份文档的资料库，从中选出两到三份与你的需求最相关的文档？即便有能够影响所有电子设备的电磁干扰，存储或传输了数百万条信息，也没犯一点错误？你是否成功地完成了一次在线交易，即便同时有成千上万名消费者在访问同一个服务器？你是否在能够被其他数十台计算机嗅探到的线路中传输了一些机密信息（比如信用卡卡号）？你是否运用过压缩的魔力，将数兆的照片压缩成更易于管理的大小，以便在电子邮件中发送？你是否在手持设备上触发了人工智能，自动纠正你在手持设备的小巧键盘上输入的内容？

这些令人印象深刻的壮举都依赖于之前提到的伟大发现。然而，绝大多数计算机用户每天都会多次运用这些独创想法，却从没有意识

到！本书旨在面向大众解释这些概念——我们每天使用的计算机科学的伟大思想。在解释每个概念时，我都假设读者不了解任何计算机科学的任何知识。

## 算法：指尖精灵的构件

到目前为止，我一直在谈计算机科学的伟大“思想”，但计算机科学家们将许多重要思想形容为“算法”。那么思想和算法之间有什么区别呢？究竟什么是算法？这一问题最简单的答案是，算法是一张精确的处方，按顺序详细列出了解决一个问题所需要的具体步骤。我们小时候在学校学到的一种算法就是很好的例子：将两个大数字相加的算法。如下例所示。这个算法涉及一连串步骤，开始的步骤如下：“首先，将两个数的最末位数相加，写下结果的最末位数，将剩下的数放到左侧的下一栏；接着，将下一栏的数相加，再将除结果末位数之外的数字和前一栏余下的数相加……”。依此类推。




将两个数字相加的算法的前两步。

请注意算法步骤近乎机械化的感觉。事实上，这是算法的关键特点之一：每一步都必须绝对精确，没有任何人类意图或推测掺杂其中。这样，每一个完全机械化的步骤才能被编入计算机。算法的另一个重要特点是，不管输入什么，算法总能运行。我们在学校学到的相加算法就拥有这一特性：不管你想把哪两个数相加，算法最终都会得出正确答案。比如，用这一算法将两个长达1 000位的数相加，你肯定能得到答案，尽管这需要相当长的时间。

对于把算法定义为一张精确、机械化的处方的说法，你也许会略感好奇。这张处方究竟要有多精确？要进行哪些基本操作？比如，在上面的相加算法中，简单地说一句“把两个数相加”是不是就行了？还是说我们要在加法表上列出所有个位数字？这些细节看起来也许有点乏味，甚至会显得有点学究气，但其实离真相不远了：这些问题的真正答案正处于计算机科学的核心，并且也和哲学、物理学、神经科学以及遗传学有联系。有关算法究竟是什么的深层问题都归结于一个前提——也就是众所周知的邱奇—图灵论题（Church–Turing Thesis）。我们将在第十章重温这些问题，届时我们还将讨论计算的理论极限，以及邱奇—图灵论题的一些方面。同时，将算法比作一张非常精确的处方这一非正式概念效果会非常好。

现在我们知道了算法是什么，但算法和计算机有什么联系呢？关键在于，计算机需要用非常精确的指令编程。因此，在能让计算机为我们解决某个特定问题之前，我们需要为那个问题开发一个算法。在数学和物理学等其他学科中，重要的结果通常是由一个方程式获得的。（著名的例子包括勾股定理$a^2+b^2=c^2$，或爱因斯坦的质量守恒定理$E = mc^2$。）相反，计算机科学的伟大思想通常是形容如何解决一个问题——当然，是使用一种算法。因此，本书的主要目的是，解释让计算机成为你的个人精灵的东西——计算机每天使用的伟大算法。

## 一个伟大的算法由什么构成？

这会引出一个刁钻的问题：什么才是真正伟大的“算法”？潜在的候选算法清单相当大，但我用几条基本标准缩减了用于本书的候选算法清单。第一条，也是最重要的一条标准是，伟大的算法要被普通计算机用户每天用到。第二条重要的标准是，伟大的算法应该能处理具体的现实问题，如压缩一个特定文件或通过一个噪声链接精确地传输

文件。对于已经了解部分计算机科学的读者而言，下面的文字框解释前面两大标准的部分后果。

第一条标准——要被普通计算机用户每天用到——排除了主要由计算机专业人士使用的算法，如编译器和程序验证技术。第二条标准——针对某个特定问题的具体程序——排除了许多作为计算机科学本科课程核心内容的伟大算法，如排序算法（快速排序）、图形算法（迪杰斯特拉最短路径算法）、数据结构（哈希表）。这些算法的伟大性毋庸置疑，而且很轻易地就满足了第一条标准，因为普通用户使用的绝大多数应用程序都会反复应用这些算法。但这些算法太通用了：它们能用于解决众多问题。在本书中，我决定要专注于解决特定问题的算法，因为对于普通计算机用户而言，这些算法能让他们拥有更清晰的动机。

一些和本书选取算法有关的额外细节。本书读者无须具备计算机科学的任何知识。但如果读者具备计算机科学背景知识，这个文字框会解释为何这类读者之前偏好的许多内容没有出现在本书中。

第三个标准是，算法主要和计算机科学理论相关。这排除了主要和计算机硬件——如CPU、监视器以及网络——有关的技术。这条标准也减轻了对基础设施——如互联网——设计的重视。为什么我要着重于计算机科学理论？部分原因是由于公众对计算机科学认知的不平衡：有一种广泛的观点认为，计算机科学基本上就是编程（如“软件”）和设备设计（如“硬件”）。事实上，最优美的计算机科学思想中有许多是十分抽象的，并不属于以上任意一类。我希望通过着重于这些理论思想，让更多人将计算机科学的本质作为一门知识学科来理解。

你也许已经注意到了，我列出的标准可能会遗漏一些伟大的算法，但却从一开始就避免了定义伟大这个极其麻烦的问题。针对这一

问题，我依赖于自己的直觉。在本书中说明的每一个算法中，其核心都是一个让整件事情奏效的精巧把戏。对我而言，当这个把戏显露出来时，那个“惊叹”时刻，会让解释这些算法成为令人愉悦的经历，我希望你也能有此感受。因为我会用到“把戏”（trick）这个词很多次，需要指出的是，我并非指那些卑劣或骗人的把戏——那种孩子可能会用在弟弟或妹妹身上的把戏。相反，本书中的把戏类似于交易诀窍或魔术：为达成目标而采用的聪明技巧，否则目标很难或不可能达成。

因此，根据直觉，我选出了自认为是计算机科学世界中最精巧、最神奇的把戏。在英国数学家高德菲·哈罗德·哈代（G. H. Hardy）的《一个数学家的辩白》（A Mathematician's Apology）中，作者试图向公众解释数学家从事数学的原因：“美是第一道测试：丑陋的数学在这个世界中无永存之地。”这道美的测试也适用于计算机科学中蕴含的理论思想。因此，选取在本书中出现的算法的最后一条标准，就是哈代的——也许可以这么称呼——美的测试：希望我至少能成功地向读者展示部分美——我在每个算法中感觉到的美。

接下来谈谈我选择展示的这些算法。搜索引擎的巨大影响，也许是算法技术影响所有计算机用户最明显的例子，我自然也将部分互联网搜索的核心算法收入了本书中。第二章描述了搜索引擎如何使用索引寻找与请求的文件，而第三章则解释了网页排名（PageRank）算法——谷歌公司为保证匹配度最高的文件出现在搜索结果列表顶部的原始算法。

即便我们不经常想这件事情，绝大多数人也能意识得到，为提供出人意料的强大搜索结果，搜索引擎使用着一些深邃的计算机科学思想。相反，其他一些伟大的算法也经常被用到，但计算机用户对此甚至都没有意识到。第四章描述的公钥加密（public key cryptography）就是这样一种算法。用户每次访问一个安全网站（地址以https而非http开头），用户都会用到公钥加密的一个方面——也就是众所周知的密

钥交换（key exchange）——来展开一段安全对话。第四章解释了密钥交换过程的实现原理。

第五章的主题是纠错码（error correcting codes），这是我们经常使用但却没有意识到的另一类算法。事实上，纠错码极有可能是有史以来唯一一个使用次数最频繁的伟大算法。纠错码可以让计算机识别并纠正在储存或传输数据中出现的错误，而不必依靠备份或再次传输。纠错码无处不在：它们被用于所有硬盘驱动器、众多网络传输、CD和DVD，甚至还存在于一些计算机的内存。不过，纠错码的能力太强了，以至于我们意识不到它们存在。

第六章稍微有点特殊，介绍了图形识别算法（pattern recognition algorithm）。图形识别算法也能进入伟大的计算机科学思想榜单，但却违背了第一条标准：要被普通计算机用户每天用到。图形识别属于计算机识别高度可变信息——如笔迹、讲话和人脸——的技术。事实上，在21世纪的第一个十年，绝大多数日常计算并没有用到这些技术。但在2011年，图形识别的重要性急剧增大：配备小型屏幕键盘的移动设备需要自动纠错，平板设备必须识别手写输入，而且所有这些设备（特别是智能手机）越来越趋向于语音操作。一些网站甚至使用图形识别来决定向用户展示哪种广告。另外，我对图形识别也有偏好，因为它是我的研究领域。因此，第六章描述了3种最有趣、最成功的图形识别技术：最近邻分类器（nearest-neighbor classifier）、决策树（decision tree）以及神经网络（neural network）。

第七章讨论了压缩算法。压缩算法组成了另一组使计算机变成我们指尖精灵的伟大思想。计算机用户的确会时不时地直接进行压缩，也许是为了节省磁盘空间，也许是为了缩减照片容量，以便用电子邮件寄出。不过在私底下，压缩使用的频率要更高：我们根本没有意识到，我们的下载或上传也可以通过压缩以节省带宽，而数据中心通常会压缩消费者的数据以降低成本。电子邮件提供商提供给你的5 GB空

间，经压缩后很有可能只占据电子邮件提供商5 GB空间的很小一部分。

第八章讲到了数据库中运用的一些基础算法。这一章侧重为实现一致性——指一个数据库中的关系不互相冲突——而采用的聪明技巧。没有这些精巧的技术，我们的绝大部分在线生活（包括网络购物以及通过Facebook之类的社交网站进行互动）就会消亡于众多计算机错误中。这一章解释了一致性真正的问题是什么，以及计算机科学家是如何解决这一问题的。前提是不牺牲我们所期望的在线系统拥有的高效性。

在第九章，我们会了解理论计算机科学无可争议的瑰宝之一：数字签名。乍看之下，用数字形式“签署”一份电子文档似乎不可能。你也许会想，这种签名必须由数字信息组成，而任何想要伪造签名的人都可以毫不费力地拷贝这些信息。这一悖论的解决方案，就是计算机科学取得的最令人瞩目的成就之一。

第十章采取了截然不同的视角：与其描述一个已经存在的伟大算法，我们不如去了解一个假如存在则必然会伟大的算法。不过我们会震惊地发现，这个特别伟大的算法不可能存在。这表明计算机解决问题的能力存在一些绝对极限，而我们将简单地从哲学和生物学角度探讨这一结果的应用。

第十一章我们会总结伟大算法的一些共性，花些时间畅想未来会怎样。会有更多伟大算法出现吗？或者说，我们已经发现了所有的伟大算法?

在此，不得不提前说一下本书的风格。任何科普作品都必须清楚地告知来源，但引用会破坏文本的流畅性，并让读者产生学术的感觉。由于可读性和易读性是本书的首要目标，所以本书正文不会出现引用。不过，我清楚地记录了所有来源，并在本书末尾的“来源和延伸

阅读”板块中列出，并时不时附上拓展评论。这个板块还列出了一些额外材料，以便感兴趣的读者能去寻找更多和计算机科学中伟大算法有关的东西。

既然提前说了本书的风格，我还要谈谈本书书名中采取的少量诗化。本书无疑是革命性的，但真的有九种算法吗？这一说法值得探讨，因为要取决于有多少算法被算作单独算法。让我们来算下“九”是怎么来的。除了前言和结论两章外，本书还有九章，每一章都介绍了对一种计算任务产生革命性影响的算法，例如加密、压缩、图形识别。因此，书名中的“九大算法”实际上指的是处理这九种任务的九类算法。

## 为什么我们要关注这些伟大的算法？

希望对这些迷人思想的快速总结能让你渴望深入了解它们的运行方式。不过，也许你仍然在思考：本书的终极目标是什么？让我简短地说下本书的真正目的。这本书绝不是一本问答式操作手册。在读完本书后，你不会成为计算机安全方面的专家，也不会成为人工智能或其他领域的专家。你也许能学到一些有用的技能，这倒是真的。比如：你会对如何检查“安全”网站凭证以及“已签名”软件包了解更多；你能针对不同任务在有损和无损压缩之间做出明智选择；而且通过理解搜索引擎索引和排名技术的某些方面，你能更高效地使用搜索引擎。

在读完本书后，你不会成为一名更加熟练的计算机用户。但你会更加珍视每天在所有计算设备上不停使用的思想的美。

为什么这是件好事？我用类比的方式来说明。我肯定不是一位天文学专家——事实上，我在这个项目上相当无知，我想知道更多。但

每当我注视夜空，我知道的少量天文学知识增强了我对这一经验的享受。有时，我对自己看到事物的理解，让我产生了一种满足和惊奇的感觉。希望在读完本书后，你在使用计算机时也能经常获得同样的满足和惊奇之感，这也是我殷切的希望。你将真正珍视我们时代最常见、最神秘的黑盒子：你的个人电脑，你指尖的精灵。

# 第二章　搜索引擎索引——在世界上最大的草垛中寻针

哈克，在咱俩站着的地方的下面，你拿一根钓鱼竿就可以触到我钻出来的那个洞。看看你能不能找到。

——马克·吐温，
《汤姆·索亚历险记》（Tom Sawyer）

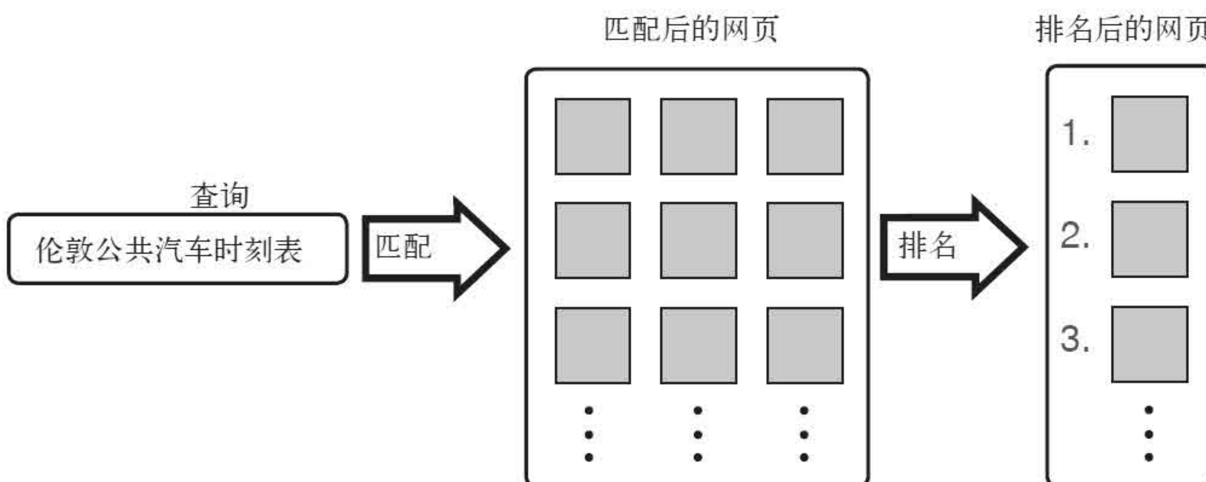
搜索引擎对我们的生活产生了深远影响。绝大多数人每天都进行多次搜索查询，但我们极少会停下来思考这个令人惊叹的工具是如何奏效的。搜索引擎提供的海量信息以及搜索结果的速度与质量变得如此平常，如果我们搜索的问题没有在几秒内得到回答，我们就会困惑。我们倾向于忘记，每一个成功的搜索引擎都是从世界上最大的草垛——万维网——寻针。

事实上，搜索引擎提供的超级服务，不仅仅是针对搜索抛出一大堆花哨技术的结果。的确，每个大型搜索引擎公司都运营着一个由无数数据中心组成的国际网络，其中包括数以千计的服务器计算机和先进的网络设备。但没有聪明的算法来组织和检索我们请求的信息，所有这些硬件都会变得毫无用处。因此，在这一章和下一章，我们将探究这样一些算法瑰宝——每次在进行网络搜索时，我们都会用到这些算法。我们很快就会了解到，搜索引擎的两大主要任务就是匹配（matching）和排名（ranking）。这一章将讲述一种聪明的匹配技

术：元词把戏（metaword trick）。在下一章，我们将转而讨论排名任务，审视谷歌公司著名的网页排名算法。

## 匹配和排名

当你发起一次网络搜索查询时会发生什么？以这样一种高屋建瓴的视角开始会很有帮助。我已经说过，搜索有两个主要阶段：匹配和排名。在实际中，搜索引擎将匹配和排名组合成一个流程以实现一致性。但这两个阶段在概念上是独立的，因此我们会假设在排名开始前，匹配已经完成。上图就给出了一个例子，图中查询的是“London bus timetable”（伦敦公共汽车时刻表），而匹配阶段则回答“哪个网页与我的查询匹配”这个问题——在这个例子中就是所有提到“London bus timetable”的网页。




网络搜索的两个阶段：匹配和排名。在第一阶段（匹配）后可能会出现数千或数百万个匹配结果，这些结果必须按照相关度在第二阶段（排名）进行排序。

但现实搜索引擎中的许多查询都有数百、数千乃至数百万个“命中”。而搜索引擎用户通常只喜欢查看几个结果，最多5个或10个。因此，搜索引擎必须从大量命中里挑出最好的几个。一个好的搜索引擎

不仅仅会挑出最好的几个命中，而且会以最有用的顺序显示它们——最匹配的页面排在第一，然后是匹配度排名第二的页面，依此类推。

以正确顺序挑选出最好的几个命中被称为“排名”。排名是关键的第二个阶段，紧随最开始的匹配阶段。在搜索行业的残酷世界中，搜索引擎的生死由其排名系统的质量决定。2002年，美国前三大搜索引擎的市场份额基本相当，谷歌、雅虎和MSN在美国的市场份额都在30%以下。[MSN随后被重新包装成Live Search，之后又被命名为必应（Bing）。]之后几年，谷歌的市场份额迅速扩大，同时将雅虎和MSN的市场份额打压到了20%以下。人们普遍认为，谷歌迅速上升为搜索行业冠军是得益于其排名算法。因此，毫不夸张地说，搜索引擎的生死由其排名系统的质量决定。不过，正如我已经提到的，我们将在下一章探讨排名算法。至于现在，让我们专注于匹配阶段吧。

## AltaVista：第一个互联网级别的匹配算法

搜索引擎匹配算法的故事从哪里开始？一个很显然却错误的回答会说从谷歌——21世纪初期最伟大的技术成功故事——开始。事实上，谷歌最初只是两位斯坦福大学研究生的博士学位项目，这个故事不仅温暖人心，而且令人印象深刻。拉里·佩奇（Larry Page）和谢尔盖·布林（Sergey Brin）在1998年组装了一堆计算机硬件来运行一种新的搜索引擎。不到10年，他们的公司成为了互联网时代崛起的最伟大的数字巨人。

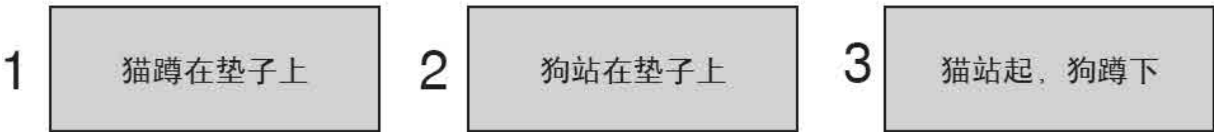
不过，互联网搜索的想法已经存在很多年了。最早的商业应用是Infoseek和Lycos（两者都于1994年推出），以及于1995年推出搜索引擎的AltaVista。20世纪90年代中期的几年中，AltaVista是搜索引擎的王者。当时我还是一名计算机科学研究生，我清楚地记得自己惊叹于AltaVista搜索结果的成熟度。有史以来第一次，有一个搜索引擎能完

全索引互联网上每一个页面的全部文本。更可贵的是，眨眼间就能返回结果。要继续理解这个令人回味的技术突破，我们要从接触一个古老的（毫不夸张）概念——索引——开始。

## 古老的索引

索引的概念是所有搜索引擎背后最基础的思想。但索引并非由搜索引擎发明：事实上，索引的思想几乎和书写本身一样古老。比如，人类学家发现了一座具有五千年历史的巴比伦神庙图书馆，里面按学科对楔形文字泥版进行了分类。因此，索引可以称得上是计算机科学中最古老的有用思想。

如今，“索引”这个词通常指参考书最后的一个板块。你可能想要查看的所有概念都以固定顺序（通常是按字母排序）列出，每一个概念下都列出了这个概念出现的位置（通常是页码）。因此，一本和动物有关的书也许会有一个像“cheetah 124，156”的索引项。这个索引项意味着“cheetah”（猎豹）这个词在第124页和第156页出现过。（让你做个相当有趣的练习，你可以在本书的索引中查询“index”这个词。你应该可以找到这一页。）




一个假想的万维网，由编号为1、2和3的三个页面组成。

| | |
|---|---|
| a | 3 |
| cat | 1 3 |
| dog | 2 3 |
| mat | 1 2 |
| on | 1 2 |
| sat | 1 3 |

| stood | 2 3 |
| --- | --- |
| the | 1 2 3 |
| while | 3 |

一个用页码表示的简单索引。

互联网搜索引擎的索引和一本书的索引有着相同的工作原理。书“页”现在成了万维网上的网页，而搜索引擎则给互联网上的每个网页分配了一个不同的页码。（是的，互联网上虽然有很多网页——最新的数据显示有成百上千亿个——但计算机很擅长处理大数字。）上图给出了一个会让整个过程更具体的例子。想象万维网只由上面显示的3个短网页组成，它们分别分配到了页码1、2和3。

计算机可以为这三个网页创建一个索引：首先要为出现在任一页面上的所有单词创建一个列表，然后按字母表顺序整理这张列表。我们可以将结果称为单词表（word list）——在这个例子中是“a、cat、dog、mat、on、sat、stood、the、while”。然后计算机会一个单词一个单词地搜遍所有页面。计算机会标注每个单词所在的页码，然后再标注单词表中下一个单词的位置。最终结果显示在上图中。比如，你可以立即看到单词“cat”出现在第1页和第3页，却不在第2页。而单词“while”只出现在第3页。

通过这种简单方法，搜索引擎就已经能回答许多简单的查询。比如，假设你输入查询词“cat”，搜索引擎能很快跳转到单词表中的“cat”项。（因为字母表是按字母排序的，计算机能很快找到任何项，就像我们可以很快找到词典中的一个单词一样。）一旦计算机找到“cat”项，搜索引擎就能给出该项的页面列表——在这个例子中就是第1页和第3页。现代搜索引擎对结果的组织很合理，只摘取了返回页面的少许片段，不过，我们基本上会忽略这样的细节，将精力集中在搜索引擎如何知道页面“符合”用户输入的查询上。

再举另一个非常简单的例子，让我们来检查一下查询“dog”的步骤。在这个例子中，搜索引擎很快会找到“dog”项，并返回页码2和3。如果查询多个单词，如“cat dog”呢？这表示你正在寻找同时包含单词“cat”和“dog”的页面。通过已有的索引，搜索引擎也能很容易查到结果。搜索引擎首先会单独查找这两个单词，找出它们分别在哪些页面中。结果是“cat”在第1页和第3页，“dog”在第2页和第3页。之后，计算机能快速扫描这两个命中列表，寻找同时出现在两个列表中的页码。在这个例子中，第1页和第2页被排除了，但第3页同时出现在两个列表中，因此最终答案就是第3页上的一次单独命中。与之极其相似的一个策略也适用于超过两个单词的查询。比如，查询“cat the sat”会返回第1页和第3页为命中，因为它们是“cat”（1，3）、“the”（1，2，3）和“sat”（1，3）这个列表的通用元素。

就目前来看，搭建一个搜索引擎听起来相当容易。最简单的索引技术似乎运行得很好，即便对多词查询也是如此。不幸的是，这种简单方法完全不能满足现代搜索引擎的需要。出现这种情况的原因有几个，不过现在我们只会关注其中之一：如何做短语查询。短语查询是指寻找一个确切短语的查询，而非凑巧一些单词出现在页面中的某些地方。比如，“cat sat”查询和cat sat查询的意义截然不同。[注]cat sat查询寻找的是在任何位置包含“cat”和“sat”两个单词的页面，不考虑顺序；而“cat sat”查询寻找的是包含单词“cat”之后紧跟单词“sat”的页面。在上面那个由三个网页组成的简单例子中，cat sat查询结果命中第1页和第3页，但“cat sat”查询只返回一次命中，就在第1页。

一个搜索引擎如何才能有效地进行一次短语查询呢？继续说“cat sat”这个例子。第一步和平常的多词查询 cat sat一样，从单词表中获取每个单词出现的网页列表，在这个例子中就是出现在第1页和第3页的“cat”；“sat”也一样，出现在第1页和第3页。不过搜索引擎到这里就卡住了。搜索引擎很确切地知道两个单词同时出现在页面1和页面3上，但没有办法来分辨这些单词是否以正确的顺序紧挨着彼此出现。

你也许会想，搜索引擎可以返回查看原网页，看这个短语是否存在。这的确是个可能的解决方案，但效率却非常非常低。这需要遍历每一个可能包含这个短语的网页的全部内容，而且可能有海量这样的网页。记住，我们在这里打交道的是一个只由三个页面组成的极小的例子，真正的搜索引擎必须从数百亿个网页中找出正确的结果。

## 词位置把戏

这一问题的解决方案是让现代搜索引擎运行良好的首个、真正精巧的思想：索引应该不单单存储页码，还要存储页面内的位置。这些位置并不神秘：它们只是代表了一个词在页面中的位置。第3个词的位置是3，第29个词的位置是29，依此类推。例子中三个页面组成的数据集如下页图所示，还加上了词位置。图下面的是索引——由存储页码和词位置中得出的结果组成。我们称这种创建索引的方法为“词位置把戏”（word location trick）。举几个例子，以确保大家理解了词位置把戏。索引的第一行是“a3-5.”。这意味着词“a”只在数据集中出现过一次，是第3页的第5个单词。索引中最长的一行是“the 1-1 1-5 2-1 2-5 3-1”。这一行可以让你知道，这个数据集中所有出现单词“the”的具体位置。它在第1页出现过两次（位置1和5），第2页出现过两次（位置1和5），第3页出现过1次（位置1）。

你还记得介绍页内词位置的目的吗？是为了解决如何有效地进行短语查询这个问题。让我们来看看如何用这个新索引做一次短语查询。还是和前面一样，查询短语“cat sat”。第一步和使用旧索引时一样：从索引中提取单个词的位置，“cat”的位置是1-2、3-2，“sat”的位置是1-3、3-7。到这里还好：我们知道短语查询“cat sat”唯一可能的命中就是在第1页和第3页。但与之前一样，我们还不确定相同的短语是否出现在了这些页面中——有可能这两个单词的确出现了，但并不是以正确的顺序彼此相邻。幸运的是，从位置信息中确认这一点很容

易。首先从第1页开始。根据索引信息，我们知道“cat”出现在第1页的位置2（这就是1-2的含义）。我们还知道“sat”出现在第1页的位置3（这是1-3的含义）。但如果“cat”在位置2，“sat”在位置3，我们就知道“sat”紧挨着出现在“cat”之后（因为2之后立即就是3）——因此我们寻找的整个短语“cat sat”必定出现在第1页，并从位置2开始。

1 the cat sat on
1 2 3 4
the mat
5 6

2 the dog stood
1 2 3
on the mat
4 5 6

3 the cat stood
1 2 3
while a dog sat
4 5 6 7

| | | | | | |
|---|---|---|---|---|---|
| a | 3-5 | | | | |
| cat | 1-2 | 3-2 | | | |
| dog | 2-2 | 3-6 | | | |
| mat | 1-6 | 2-6 | | | |
| on | 1-4 | 2-4 | | | |
| sat | 1-3 | 3-7 | | | |
| stood | 2-3 | 3-3 | | | |
| the | 1-1 | 1-5 | 2-1 | 2-5 | 3-1 |
| while | 3-4 | | | | |

底图：同时包含页码和页内词位置的新索引。
顶图：3个网页加上了页内词位置。

我知道自己在这点上谈得很多，但巨细无遗、从头至尾地研究这个例子，是为了让读者真正地理解为了获得答案究竟使用了哪些信息。注意，我们已经为短语“cat sat”找到了一次命中，仅仅是通过查看索引信息（“cat”的位置1-2、3-2，“sat”的位置1-3、3-7），而非原始网页。这很关键，因为我们只需查看索引中的两个项，而非遍历所有可能包含命中的页面——而在进行真正短语查询的真正的搜索引擎中，可能有数百万个这样的页面。总之：通过在索引中加入页内词位置，我们只需通过查看索引中的几行，就能找到一次短语查询命中，而非遍历海量页面。这个简单的词位置把戏是让搜索引擎奏效的关键之一。

事实上，我还没讲完“cat sat”这个例子。我们完成了对第1页信息的处理，但第3页还没处理。对第3页的推理和第1页的处理方式很相似：“cat”出现在第3页的位置2，“sat”出现在位置7，因此它们不可能相邻——因为紧跟2之后出现的不是7。这样我们就知道，第3页并不是短语查询“cat sat”的命中，尽管它是多词查询cat sat的命中。

顺便说一下，词位置把戏不仅仅对短语查询重要。举个例子，思考一下寻找相邻单词的问题。在一些搜索引擎中，用户可以在查询中使用“NEAR”关键词做到这一点。事实上，AltaVista搜索引擎在早期就提供了这一功能，在本书写作时仍然提供。作为一个特殊的例子，假设在一些特别的搜索引擎中，查询cat NEAR dog 会找到“dog”前后五个位置之内出现“cat”的页面。我们如何才能在数据集中有效地执行这种查询？使用词位置，会使查询变得很容易。“cat”的索引项是1-2、3-2，而“dog”的索引项是2-2、3-6。可以立刻看出，第3页是唯一可能的命中。而在第3页，“cat”出现在位置2，“dog”出现在位置6。因此这两个词之间的距离是6－2，结果是4。因此，“cat”的确出现在“dog”前后五个位置之内，而第3页则是查询cat NEAR dog的命中。和前面一样，请注意这次查询的执行是多么高效：无须遍历任何网页的实际内容——相反，只参考了索引中的两个项。

不过，在实际中，NEAR查询对搜索引擎用户并不非常重要。几乎没人使用NEAR查询，绝大多数主要搜索引擎甚至不支持它们。尽管如此，能执行NEAR查询的能力实际上对现实中的搜索引擎至关重要。这是因为搜索引擎不断地在后台执行NEAR查询。要理解其中的原因，我们首先不得不研究现代搜索引擎面临的主要问题之一：排名的问题。

## 排名和邻度

到目前为止，我们一直专注于匹配阶段：为一个给出的查询高效地找出所有命中的问题。不过正如之前强调的，第二个阶段“排名”对于一个高质量的搜索引擎是绝对必不可少的：这是挑选出前几个命中并展示给用户的阶段。

让我们更细致地来检验排名的概念。一个网页的“排名”究竟取决于什么？真正的问题不是“这个网页和查询匹配吗”，而是“这个网页和查询相关吗”。计算机科学家们使用“相关度”（relevance）这个术语来形容一个结果网页和某个特定查询有多么相配或多么有用。

举个具体的例子，假设你对导致疟疾的原因感兴趣，并在一个搜索引擎中输入查询malaria cause（导致疟疾）。简化考虑，假设搜索引擎对这一查询只有两个命中——下图显示的两个网页。现在来看看这两个网页。作为人类，你很快就知道第1页和疟疾起因有关，而第2页似乎是对刚刚发生的一些军事行动的描述，只不过恰巧使用了“cause”和“malaria”这两个词。因此，和第2页相比，第1页无疑和查询malaria cause更具相关性。可计算机不是人，让计算机理解这两页的主题也很难，似乎不可能让搜索引擎正确地对这两个命中进行排名。

1 By far the most common cause of malaria is being bitten by an infected mosquito, but there are also other ways to contract the disease.

2 Our cause was not helped by the poor health of the troops, many of whom were suffering from malaria and other tropical diseases.

| | | |
|---|---|---|
| also | 1-19 | |
| ... | | |
| cause | 1-6 | 2-2 |
| ... | | |
| malaria | 1-8 | 2-19 |
| ... | | |
| whom | 2-15 | |

顶图：两个提及疟疾的样本网页。
底图：从上方两个网页中创建的索引的一部分。

不过，事实上，有一种很简单的方法让这个例子中的排名正确。查询词彼此相邻的网页比那些查询词相距很远的网页相关度更高。在疟疾这个例子中，“malaria”和“cause”在第1页中仅相距1个词，而在第2页中则相距17个词。（记住，搜索引擎只通过查看索引项就能高效地发现这一点，无须返回查看网页。）因此，尽管计算机并不真正地“理解”查询的主题，它也能猜测网页1比网页2更具相关性，因为网页1查询词之间的距离要比网页2更近。

总而言之，尽管人们不经常使用NEAR查询，搜索引擎也在不断地使用和邻度有关的信息，提高搜索排名。而它们能高效地做到这点的原因则是，它们使用词位置把戏。

1 my cat
the cat sat on the mat

2 my dog
the dog stood on the mat

3 my pets
the cat stood while a dog sat

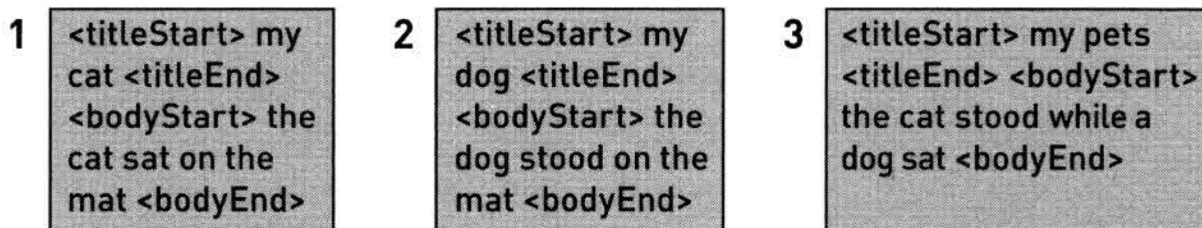
一个网页范例集，每个网页都有一个标题和一段正文。

我们已经了解到，早在距今5 000年以前，巴比伦人就开始使用索引。而词定位把戏也不是由搜索引擎发明的：这是互联网出现以前，另一种信息检索中用到的著名技术。不过，在下一部分，我们将了解一个看起来的确是由搜索引擎设计者发明的新把戏：元词把戏（metaword trick）。对这一把戏和众多相关思想的精巧运用，使AltaVista搜索引擎在20世纪90年代晚期迅速成为搜索行业的领头羊。

## 元词把戏

到目前为止，我们一直都在使用极其简单的网页示例。然而，绝大多数网页拥有众多结构，包括标题、标头、链接和图片，可我们还一直认为网页只是普通的词表。接下来，我们将探索搜索引擎如何处理网页中的结构。不过，为了尽可能保持简单，我们只会引入一层结构：网页的顶部会有一个标题，之后是页面的正文。上图显示了我们熟悉的三页示例，并附加了一些标题。

实际上，要像搜索引擎一样分析网页结构，我们需要了解更多编写网页的知识。网页是由一种特殊语言编写的，以便网络浏览器能用很好的格式展示它们。（编写网页最常用的语言被称为HTML，不过HTML的细节对本次讨论不重要。）标头、标题、链接、图片等格式化结构是用被称为元词的特殊单词编写的。比如，网页标题开始使用的元词也许是<titleStart>，而结束这个标题的元词可能是<titleEnd>。类似的，网页正文可能是以<bodyStart>开始，以<bodyEnd>结束。不要纠结于“<”、“>”这些符号。它们出现在绝大多数计算机键盘上，人们通常只知道这些符号的数学意义是“大于”和“小于”。不过在这里，这些符号和数学没有任何关系，只是方便的象征，将这些元词和网页中的正常单词区分开来。




和上图一样的网页集，但展示的是用元词编写的情况，而非在网络浏览器中显示的样子。

看一下上面的图。这张图展示的内容和前一张图一样，但显示的是实际编写网页的样子，而非在网络浏览器中显示的样子。绝大多数网络浏览器都能让用户检验网页的原始内容，这需要选择名为“查看网页源代码”的菜单选项——我建议你下次有机会试验一下。（注意，在这里使用的元词，如<titleStart>和<titleEnd>是帮助你理解的虚构的、易于辨认的示例。在真实的HTML中，元词被称作标签（tag）。HTML中开启和结束标题的标签是<title>和</title>——你可以在使用“查看网页源代码”的菜单选项后搜索这些标签。）

在创建一份索引时，囊括所有元词是件很简单的事。无须新把戏：你只要像存储正常单词一样存储元词位置就行。下页的图显示了从带有元词的三个网页中创建的索引。看一下这张图，确保自己理解了其中所有的奥秘。比如，“mat”的项是1-11、2-11，表示“mat”是第1页的第11个词，也是第2页的第11个词。元词位置的解读也一样，“<titleEnd>”的项是1-4、2-4和3-4，也就是说“<titleEnd>”是第1页、第2页和第3页的第4个词。

我们称这种和索引普通单词一样索引元词的简单把戏为“元词把戏”。这个把戏也许看起来简单得可笑，但元词把戏在让搜索引擎执行精确搜索和高质量排名上扮演了至关重要的角色。举个简单例子证明。假设在某个时候，有个搜索引擎支持使用IN关键词的特殊查询，因此像boat IN TITLE这样的查询只会返回在网页标题中包含了单词“boat”的网页，而查询giraffe IN BODY则会找到在正文中包含“giraffe”（长颈鹿）的网页。请注意，绝大多数搜索引擎并不完全按

照这种方法提供IN查询，但一些搜索引擎可以通过让你点击“高级搜索”选项，详细说明查询词必须出现

| | |
|---|---|
| a | 3-10 |
| cat | 1-3 1-7 3-7 |
| dog | 2-3 2-7 3-11 |
| mat | 1-11 2-11 |
| my | 1-2 2-2 3-2 |
| on | 1-9 2-9 |
| pets | 3-3 |
| sat | 1-8 3-12 |
| stood | 2-8 3-8 |
| the | 1-6 1-10 2-6 2-10 3-6 |
| while | 3-9 |
| <bodyEnd> | 1-12 2-12 3-13 |
| <bodyStart> | 1-5 2-5 3-5 |
| <titleEnd> | 1-4 2-4 3-4 |
| <titleStart> | 1-1 2-1 3-1 |

上一张图中的网页（包括元词）的索引。

dog : 2-3 2-7 3-11

<titleStart> : 1-1 2-1 3-1

<titleEnd> : 1-4 2-4 3-4

搜索引擎如何有效地执行查询dog IN TITLE。

在标题或一份文件的特定位置来实现同样的效果。我们假定IN关键词存在，以便更容易解释。事实上，在写作本书时，谷歌已经可以让用户通过使用关键词intitle进行标题搜索：因此，在谷歌中查询intitle:boat，将找到标题中带有“boat”的网页。自己试试！

让我们来看看，在上面两张图中由三个网页组成的示例里。

首先，搜索引擎提取“dog”的索引项，也就是2-3、2-7和3-11。然后（这可能有点出人意料，但请忍耐片刻）搜索引擎会同时提取<titleStart>和<titleEnd>的索引项。

<titleStart>的索引项是1-1、2-1和3-1，<titleEnd>的索引项是1-4、2-4和3-4。这些提取信息全部显示在上图中，你可以忽略那些圈和框。

之后，搜索引擎开始扫描“dog”的索引项，检查其命中，看是否有哪个命中发生在标题内。“dog”的第一个命中是圈起来的项2-3，代表其是第2页的第3个词。通过一并扫描<titleStart>的项，搜索引擎就能知道第2页的标题从哪开始——即索引项的第一个数字要以“2-”开始，也就是被圈的项2-1，即第2页的标题从第1个单词处开始。同样的，搜索引擎能知道第2页的标题在哪结束。搜索引擎只要扫描索引项中的<titleEnd>，寻找以“2-”开始的索引项数字，在这个例子中就是停止在被圈的项2-4处。因此第2页的标题在第4个词处结束。

我们目前已知的所有东西都被图中圈住的项总结了。它们告诉我们第2页的标题从第1个词开始，到第4个词结束，而“dog”这个词是第3个词。最后一步很简单：因为3大于1，小于4，我们肯定“dog”的这次命中确实出现在一个标题中，因此第2页应该是查询dog IN TITLE的命中。

现在搜索引擎可以转向寻找“dog”的第二个命中，也就是2-7（第2页的第7个词），但因为我们已经知道第2页是命中，因此可以忽略2-7这个项，转向下一个命中3-11（由一个框标记）。这表示“dog”是第3页第11个词。于是我们开始跳过被圈住的<titleStart>和<titleEnd>项，寻找以“3-”开始的项。（有一点需要重点注意，我们不必回到每行的开始，而是可以从之前扫描命中的地方重新开始。）在这个简单例子中，以“3-”开始的项恰好彼此相邻——<titleStart>是3-1，<titleEnd>是3-4。为便于参考，这两个数字都用框围了起来。接下来，我们又面临判定“dog”在3-11的命中是否位于标题内的问题。框内信息告诉我们，它们都是在第三页，“dog”是第11个词，而标题从第1个词开始，到第4个词结束。因为11大于4，所以“dog”的这次命中出现在标题之后，也就是不在标题内。网页3并不是查询dog IN TITLE的命中。

元词把戏能让搜索引擎以极端高效的方式回应有关一个文件结构的查询。上面的例子只是搜索页面标题内，但类似的技术能让用户搜索超链接、图片描述和网页其他有用部分内的词。而且所有这类查询都可以像上面的例子一样得到高效回应。正如我们之前讨论过的查询，搜索引擎无须返回查看原始网页：搜索引擎只需查阅小部分索引项，就能回应查询。同样重要的是，搜索引擎只需遍历每个索引项一次。还记得我们在完成处理第2页的首个命中后，转向第3页的可能命中时发生了什么吗？搜索引擎并没有返回索引项<titleStart>和<titleEnd>的开端，而是从之前离开的地方继续进行扫描。这也是让IN查询高效的关键因素。

标题查询和其他取决于网页结构的“结构查询”类似于之前讨论的NEAR查询，虽然人们极少执行结构查询，但搜索引擎无时无刻不在内部使用它们。原因之前提过：搜索引擎的生死由其排名的质量决定，而通过利用网页结构，排名质量能够得到大幅提升。比如，标题中有“dog”的网页包含与狗有关信息的可能性，要比在网页正文中提及“dog”的网页大得多。因此，当一名用户输入简单的查询dog，搜索

引擎能在内部执行一个dog IN TITLE查询（即便用户并未详细地要求这一点），以寻找最有可能与狗有关的网页，而非只是恰好提到狗的网页。

## 索引和匹配把戏并非是全部内容

搭建一个搜索引擎并不是一件容易的事情。最终成品就像一个巨大的复杂机器，带有许多不同的轮子、发动机和杠杆。这些装置都必须安装正确，系统才能有用。因此，单靠在本章中出现的两个把戏并不能解决创建一个高效搜索引擎索引的问题，意识到这一点很重要。不过，词位置把戏和元词把戏无疑展现了真正的搜索引擎构建和使用索引的“风味”。

元词把戏的确帮助过AltaVista——其他搜索引擎则失败了——成功地在整个互联网中寻找有效匹配。我们之所以知道这一点，是因为AltaVista在1999年递交的美国专利文件《索引的限制搜索》（Constrained Searching of an Index）中描述了元词把戏。不过，AltaVista超级精巧的匹配算法并不足以让其从搜索行业波涛汹涌的早期脱颖而出。正如我们已经知道的，有效匹配只是一个高效搜索引擎的一半，另一大挑战是对匹配网页进行排名。正如我们将在下一章中看到的，一种新排名算法的出现足以让AltaVista相形见绌，并让谷歌一跃进入网络搜索世界的最前沿。

---

1. 注意英文单词的双引号。——译者注

# 第三章　PageRank——让谷歌腾飞的技术

《星际迷航》（Star Trek）中的计算机并不特别让人感兴趣。他们向计算机提问题，计算机还要想一会儿。我觉得我们能做得更好。

——拉里·佩奇（谷歌联合创始人）

从建筑学的角度来说，车库基本上是个简陋的地方。但在硅谷，车库有一种特殊的创业含义：许多伟大的硅谷技术公司在此诞生或至少从车库中孵化而来。这一趋势并非从20世纪90年代的互联网泡沫开始。在互联网泡沫出现的50多年前，也就是1939年，当世界经济仍未从大萧条的影响中走出来时，惠普（Hewlett-Packard）就在加利福尼亚州帕洛阿尔托（Palo Alto）戴夫·休利特（Dave Hewlett）的车库中逐渐成形了。几十年之后，史蒂夫·乔布斯（Steve Jobs）和史蒂夫·沃兹尼亚克（Steve Wozniak）于1976年在加利福尼亚州洛斯拉图斯乔布斯的车库中创业，之后创建了今天传奇的苹果计算机公司。（尽管传说苹果公司创办于车库，乔布斯和沃兹尼亚克一开始其实是从一间卧室开始的。空间很快就不够用了，于是他们转移到了车库。）不过，和惠普和苹果的成功故事相比，一个名为谷歌的搜索引擎的创办过程更令人惊叹。谷歌从加利福尼亚州门洛帕克市的一间车库开始，并于1998年9月注册成立公司。

那时，谷歌事实上已经运营自己的搜索引擎一年多了——最开始是在斯坦福大学的服务器上，谷歌的两位联合创始人都是斯坦福博士

生。直到斯坦福大学再也不能承受这一日益受欢迎的服务所需要的带宽，拉里·佩奇和谢尔盖·布林才把公司转移到了如今著名的门洛帕克车库。他们肯定做了些正确的事，因为在他们正式成立公司3个月后，美国《个人计算机杂志》（PC Magazine）就宣布谷歌是1998年美国排名前一百的网站之一。

这也是我们的故事真正开始的地方：在当年《个人计算机杂志》的评论中，谷歌的精英管理层因为谷歌“以超乎寻常的技巧返回相关度极高的结果”而获奖。你也许还记得上一章提到过，第一个商业搜索引擎于4年前的1994年发布。还在车库里的谷歌怎么能弥补4年的巨大差距，在搜索质量上超越已经很受欢迎的Lycos和AltaVista呢？这一问题的答案可不简单。但最重要的因素之一——尤其是在网络搜索早期——就是谷歌用来对其搜索结果进行排名的创新算法：一个被称为PageRank的著名算法。

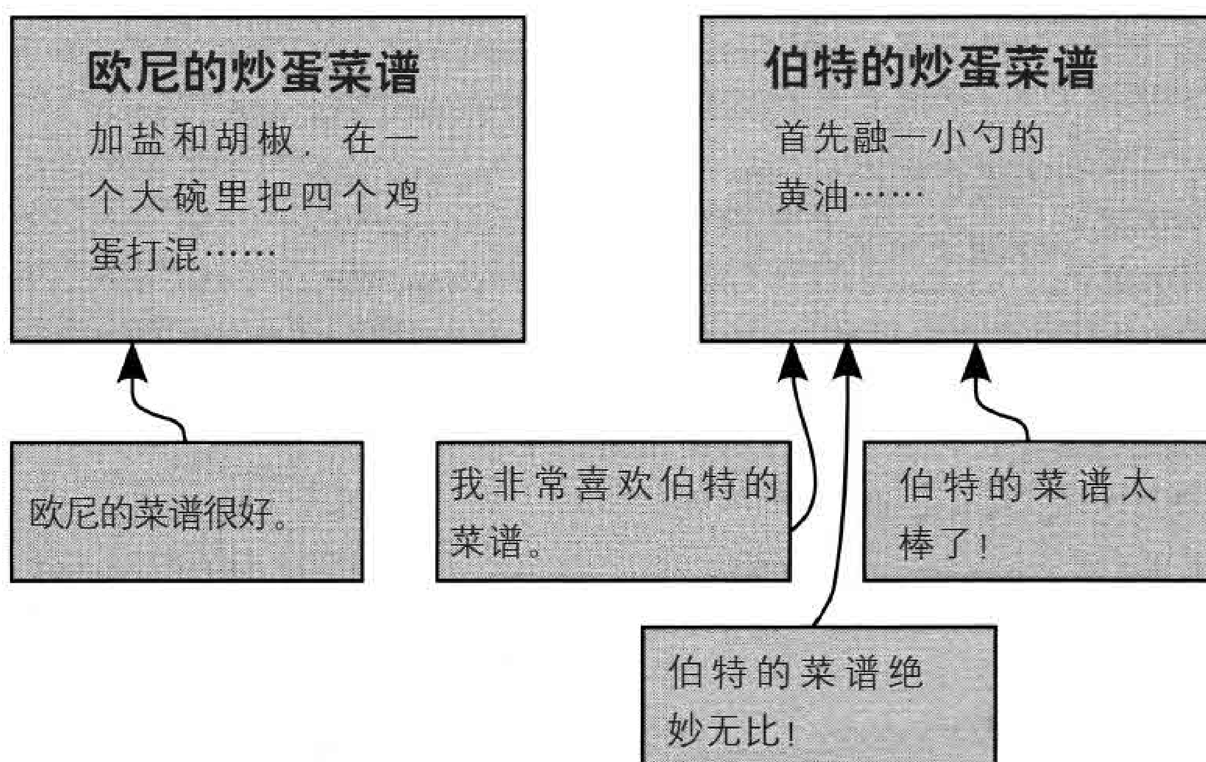
“PageRank”是个双关词：它既是一种对网页排名的算法，也是其主要发明者拉里·佩奇的排名算法。佩奇和布林在1998年的一篇学术会议论文《解析大规模超文本网络搜索引擎》（the Anatomy of a Large-Scale Hypertextual Web Search Engine）中发表了这一算法。正如论文标题所暗示的，这篇论文的内容不止是描述PageRank。事实上，这是对1998年存在的谷歌系统的完整描述。但藏在这一系统技术细节中的，是对也许是21世纪出现的第一个算法瑰宝的描述：PageRank算法。在本章，我们将探索这一算法如何以及为什么能在草垛中寻针，并持续为搜索查询提供最相关的结果——也是排名最靠前的命中。

## 超链接把戏

你很有可能已经知道了超链接是什么：超链接是网页上的一个短语，当你点击它时，你将被带到另一个网页。绝大多数网络浏览器用

蓝色底线显示超链接，以便能轻易识别。

令人意外的是，超链接也是老想法。1945年——大约在同时开始开发电子计算机——美国工程师范内瓦·布什（Vannevar Bush）发表了一篇极具前瞻性的论文《诚若所思》。在这篇涉猎广泛的论文中，布什描述了大量可能的新技术，包括一台被称作麦麦克斯（memex）的机器。麦麦克斯可以存储文件并自动进行索引，但其功能远不止这些。麦麦克斯允许“关联索引……任何被选中的东西都能立即自动选择另一个东西”——换句话说，一种早期的超链接。




超链接把戏（the hyperlink trick）的原理。上面显示了6个网页，每个框都代表1个网页。其中2个网页是炒蛋菜谱，其余4个网页都有这些菜谱的超链接。超链接把戏认为伯特的网页比欧尼的网页排名高，因为伯特有三个链入链接（incoming link），而欧尼的只有一个。

超链接自1945年就已出现。它们是搜索引擎用来进行排名最重要的工具之一，而且是谷歌PageRank技术的基础。接下来，我们将开始以最大的热情探索PageRank技术。

理解PageRank的第一步是一个名为超链接把戏的简单想法。用一个例子就能非常容易地解释这个把戏。假设你对学习如何制作炒蛋感兴趣，并且用网络搜索了这一主题。如今，任何一次真正搜索炒蛋的网络搜索都会出现数百万个命中，但为方便起见，让我们想象只有两个网页出现：其中一个是“欧尼的炒蛋菜谱”，而另一个则是“伯特的炒蛋菜谱”。这两个网页都出现在上图中，与之一道的是拥有这些菜谱超链接的网页。还是为了方便起见，让我们想象这四个包含超链接的网页是整个互联网上仅有的链接到两个菜谱网页之一的网页。图中底部画线的文字就代表超链接，而箭头则表示链接的指向。

问题是，这两个命中哪个排名应该更高？伯特还是欧尼？人们在阅读链向这两份菜谱的网页并作出评价上不会有太大的问题。看起来这两份菜谱都很合理，但人们对伯特菜谱的反响要更为热烈一些。因此，在没有给出其他信息的情况下，伯特的菜谱比欧尼的菜谱排名更高可能会更合理。

不幸的是，计算机并不擅长理解网页的真实意思，因此搜索引擎检查这四个链向命中的网页，并对每份菜谱获推荐的强烈程度进行评估也不太可能。另外，计算机在计算方面非常优秀。一种简单方法就是只计算链向每份菜谱的网页数——在这个例子中，一个网页链向欧尼的菜谱，三个网页链向伯特的菜谱——并根据这些菜谱的链入链接数对菜谱排名。当然，这种方法远不如让人阅读所有页面并手动排名精确，但无疑是一种有用的方法。如果你没有其他信息，一个网页的链入链接数可以成为该网页可能会多有用或多有“权威性”的指标。在这个例子中，伯特的菜谱得分为3，欧尼的菜谱得分为1，因此在搜索引擎向用户展示的结果中，伯特的网页排名比欧尼的高。

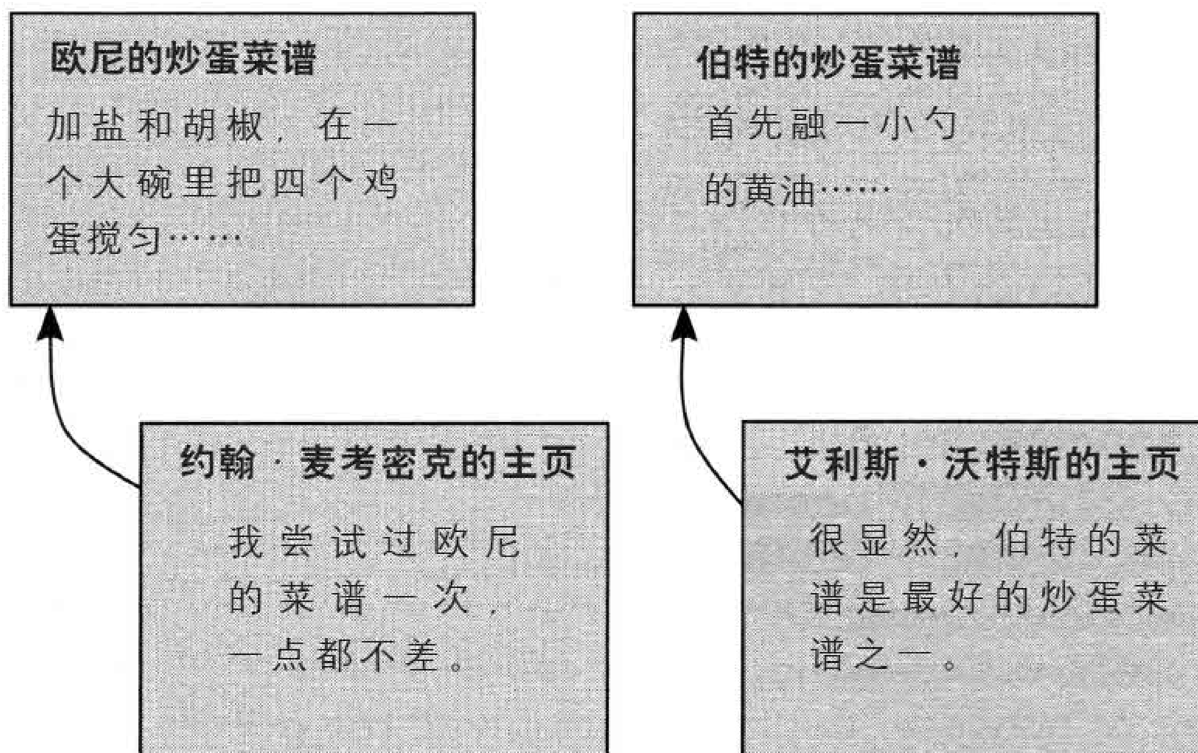
你可能已经发现了一些在排名上使用这种“超链接把戏”的问题。一个很明显的问题就是，有时候链接被用来显示差网页，而非好网页。比如，假设有个链接欧尼菜谱的网页上写着：“我试了下欧尼的菜

谱，很糟糕。”像这样批评而非推荐一个网页的链接，的确会导致超链接把戏将网页的排名拔高。不过，在现实中，超链接更多是用于推荐而非批评。因此，尽管有这个明显的缺陷，超链接把戏仍然很有用。

## 权重把戏

你可能已经在想，为什么要对网页的所有链入链接一视同仁。来自专家的推荐肯定就要比菜鸟的推荐更有价值？要细致地理解这一点，我们继续研究上面的炒蛋例子，不过研究的是另一组链入链接。下页的图对链入链接进行了重新设置：现在，伯特和欧尼的菜谱的链入链接数相等了（只有一个），但欧尼的链入链接来自我的主页，而伯特的则来自于著名主厨艾利斯·沃特斯。

如果没有其他信息，你更喜欢哪个菜谱？很显然，选择由一位著名主厨推荐的菜谱，要比选择由一名计算机科学相关书籍作者推荐的菜谱更好。我们称这一基本原则为“权重把戏”（the authority trick）：来自高“权重”网页的链接排名要比来自低“权重”网页链接的排名高。

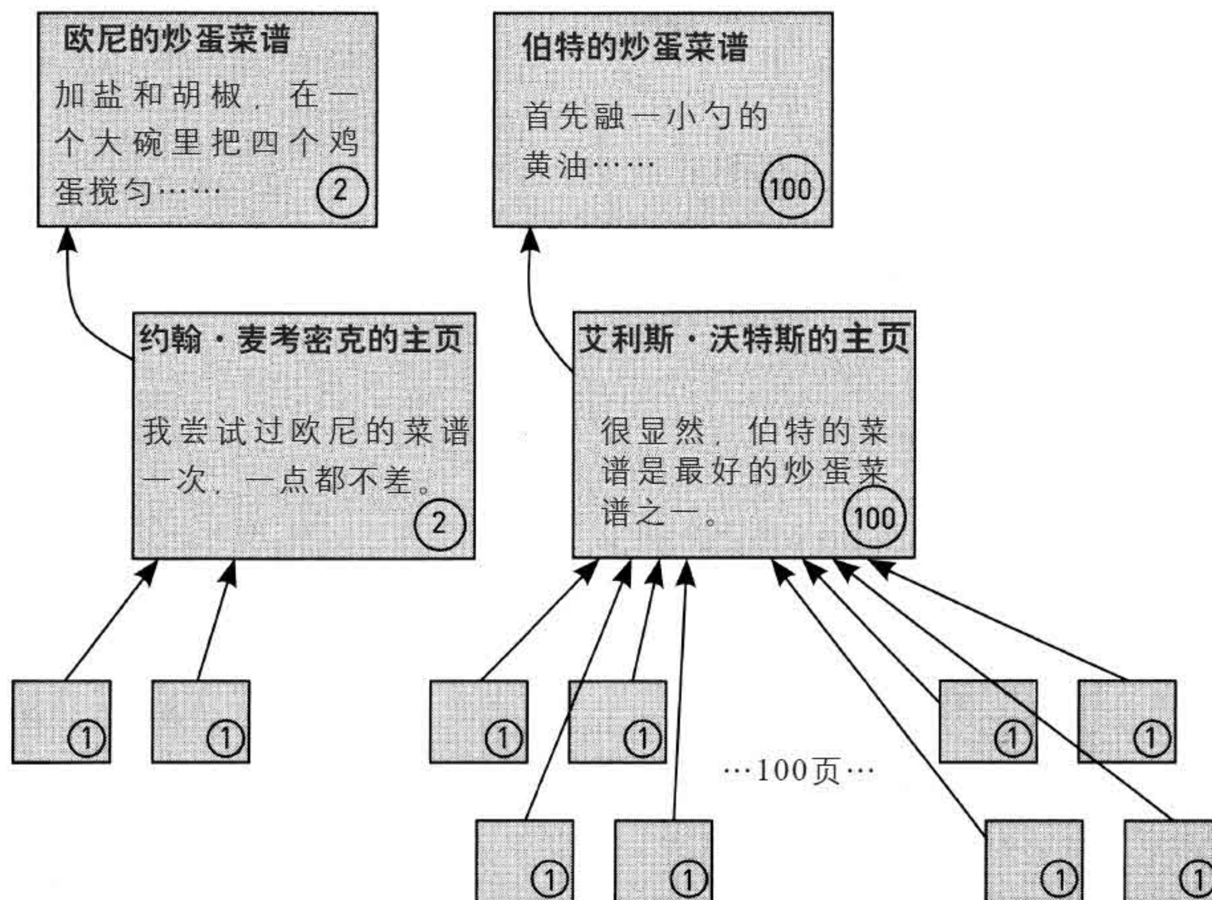



权重把戏的原理。这里显示了四个网页：两个炒蛋菜谱网页和两个链向菜谱的网页。其中一个链接来自于本书作者（不是著名主厨），而另一个链接来自著名主厨艾利斯•沃特斯的主页。权重把戏将伯特的网页排在欧尼的菜谱之前，因为伯特的链入链接“权重”比欧尼的链入链接大。

这个原则很好，但其实际形式对搜索引擎而言一点用都没有。计算机如何才能自动判定艾利斯·沃特斯在炒蛋方面比我更具有权威性呢？有个想法对此也许会有所帮助：让我们把超链接把戏和权重把戏结合起来。所有网页的初始权重值（authority score）都是1，但如果一个网页有链入链接，在计算该网页权重时就要加入指向其的网页的权重。也就是说，如果X和Y网页链向Z网页，那么Z网页的权重就是X网页和Y网页权重相加的值。

下面的图在计算这两个炒蛋菜谱网页的权重值上很详细。终值显示在圆圈中。图中有两个网页链向我的主页；这些网页本身没有链入链接，因此权重值为1。我的主页的权重值是所有链入链接权重值的总和，相加得2。艾利斯·沃特斯的主页有100个链入链接，每个链入链接

的权重值为1，因此它的权重是100。欧尼的菜谱只有一个链入链接，但这个链入链接的权重值是2，因此将其所有链入链接的权重值相加（这个例子中只有一个数可加），欧尼菜谱网页的权重值为2。伯特菜谱网页也只有一个链入链接，但其权重值为100，因此伯特菜谱网页的权重值为100。而因为100大于2，所以伯特的网页排名要比欧尼的高。

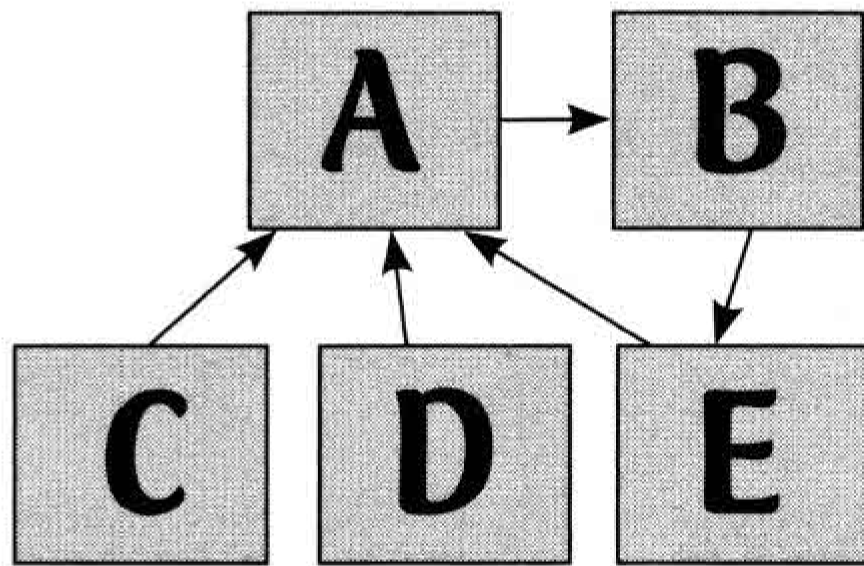



对两个炒蛋菜谱网页“权重值”的简单计算。权重值显示在圆圈中。

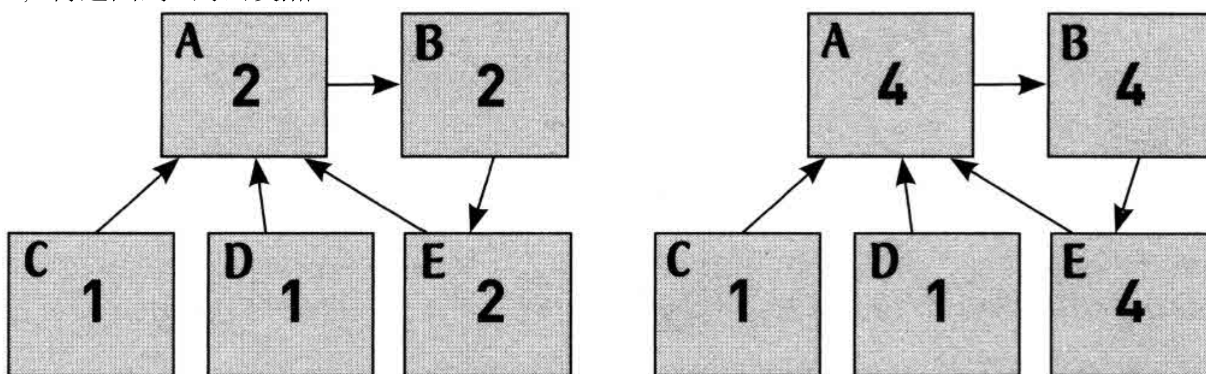
## 随机访问者把戏

就自动计算权重值来说，我们似乎拥有了一个真正奏效的策略，无须计算机真正地理解网页内容。不幸的是，这种方法有个大问题。超链接很有可能形成被计算机科学家称为“循环”（cycle）的东西。循环指访问者可以通过点击超链接返回出发时的网页。

下图就是个例子。图中有A、B、C、D、E五个网页。如果从A开始，我们可以通过A访问B，然后又从B访问E，而从E我们又能点回A，也就是回到了出发点。这也意味着A、B和E三个网页组成了一个循环。




超链接循环的一个例子。网页A、B和E组成了一个循环，你可以从A开始，点击到B，然后到E，再返回到A的出发点。

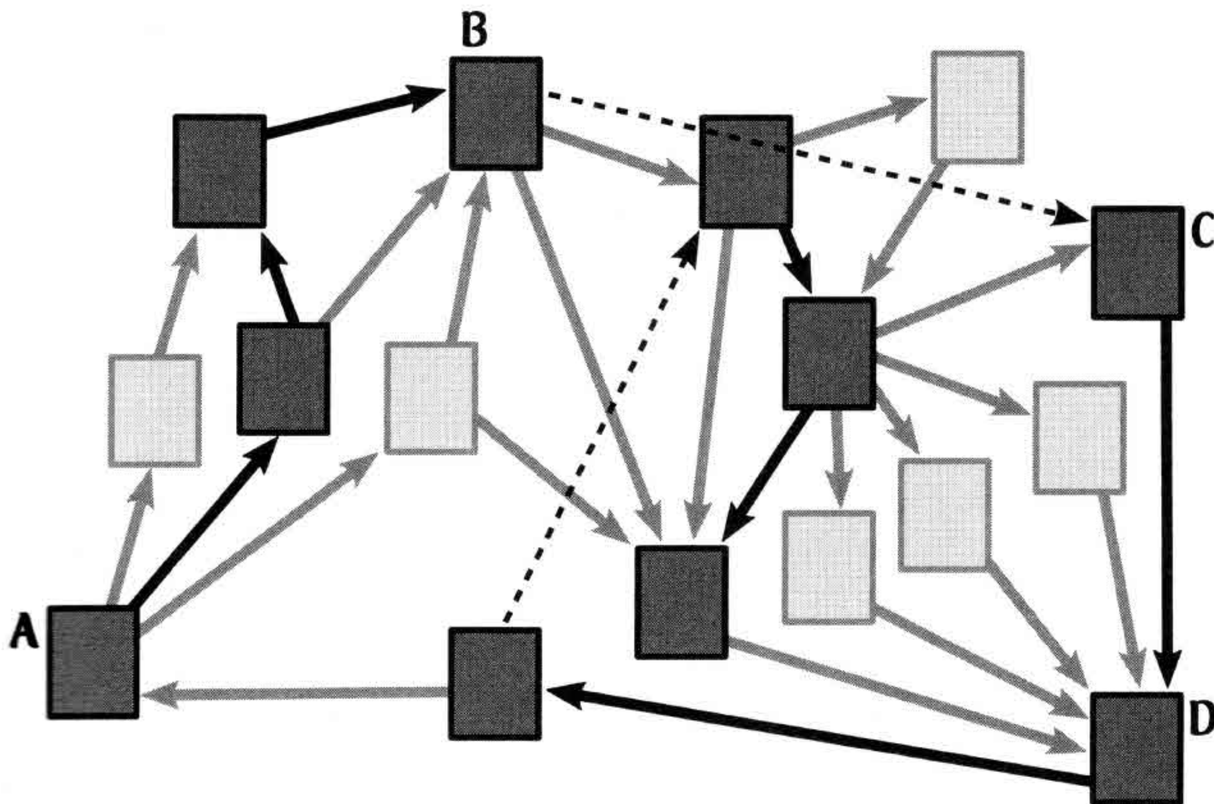



循环造成的问题。网页A、B和E永远需要更新，它们的权重值会一直增加下去。

看来，在遇到循环时，目前“权重值”的定义（将超链接把戏和权重把戏结合起来）就碰到大麻烦了。看看在这个特例中会发生什么事情。网页C和D没有链入链接，因此其权重值为1。网页C和D都链向网页A，因此A的权重值是网页C和D权重值的和，也就是1+1=2。B网页

从A获得的权重值为2，而E又从B获得权重值2。（这里谈到的情况都由上图左侧部分所体现。）但现在A的权重值要更新了：A从C和D各得到权重值1，但也从E得到权重值2，相加为4。于是B的权重值也需要更新：从A获得权重值4。然后E的权重值也要更新，它从B获得了权重值4。（现在谈到的情况都由上图右侧部分所体现。）依此类推：于是A的权重值变为6，B为6，E为6，于是A的权重值变为8……你懂了吧？随着循环进行，网页的权重值会一直增加。

这样计算权重值，会产生“鸡生蛋，还是蛋生鸡”的问题。如果我们知道A网页真正的权重值，我们就能计算B网页和E网页的权重值。而如果我们知道B网页和E网页真正的权重值，我们就能计算A网页的权重值。但由于这些网页彼此依赖，似乎这样计算根本行不通。




随机访问者模式，被访问者访问的网页用灰色表示，虚线箭头代表随机重新开始访问（restart),实线箭头从网页A开始，指向随机选择的超链接，并被两个随机重新开始访问箭头所打乱。

幸运的是，我们可以通过被称为随机访问者把戏（the random surfer trick）的技术解决这个“鸡生蛋，还是蛋生鸡”的问题。注意：对随机访问者把戏的初始描述，和到目前为止探讨的超链接及权重把戏没有任何关联。一旦了解了随机访问者把戏的基本原理，我们就会做一些分析，揭示其令人瞩目的品质：随机访问者把戏结合了超链接及权重把戏令人喜爱的属性，但在出现超链接循环时也行得通。

这个把戏从假想一个随机访问互联网的人开始。确切地说，访问者随机从万维网上的一个网页开始访问，然后检查该网页上的所有超链接，之后随机挑选出其中一个超链接进行点击。用户再检查打开的新网页，随机选择一个超链接打开。这个过程会持续进行，每一个网页都是通过随机选择前一个网页上的链接打开的。上图就是个例子。在这个例子中，我们假设整个万维网只有16个网页。框代表网页，箭头代表网页之间的超链接。其中标记了四个网页以便稍后进行参考。被访问者访问的网页用灰色表示，黑色箭头代表访问者点击的超链接，虚线箭头代表随机重新开始访问，这个在之后会讲到。

整个过程有一个转折点：每次访问一个网页时，都有一个固定的重新访问概率（大概是15%），让访问者不从已有的超链接中挑选一个并点击。相反，访问者会重新开始这一过程，从互联网上随机选择一个网页点击。你也可以认为访问者有15%的概率对任何已有网页厌倦，导致其点击另一组链接，这么想也许会有帮助。要想找些例子，请仔细观察上图。这个特定的访问者从网页A开始，在对网页B厌倦前连续点击了三个随机超链接，并在网页C重新开始。在下次重新开始前，访问者又点击了两个随机超链接。（顺便说一句，本章中所有随机访问者例子中的重新开始概率都为15%，这也是谷歌联合创始人拉里·佩奇和谢尔盖·布林在描述其搜索引擎原型的原始论文中使用的值。）

用计算机模拟这一过程很容易。我为此写了一个程序并运行了它，直到访问者访问了1 000个网页。（当然，这并不意味着是1 000个不重复的网页。对同一网页的多次访问也被纳入了计算当中，在这个例子中，所有网页都被访问了多次。）这1 000次模拟访问的结果显示在下图（顶图）中。你可以看到，网页D的访问次数最多，有144次。

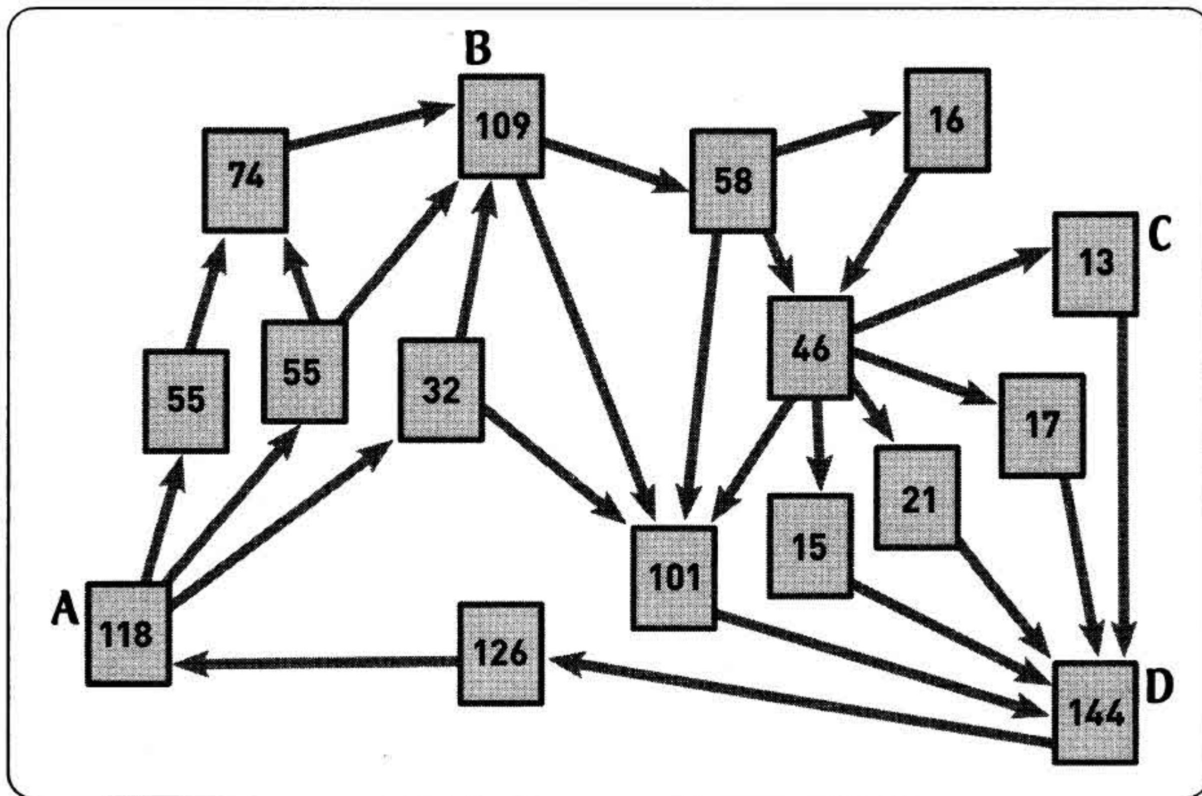
就像民意调查一样，我们可以通过增加随机样本的数目来提高模拟精度。我重新运行了一次模拟，直到访问者访问了一百万个网页。（也许你会想这花了多长时间，在我电脑上运行只花了不到半秒！）考虑到访问量如此巨大，还是用百分比表示结果更好。这也就是你将在下图（底图）中看到的情形。和之前的结果一样，网页D的访问次数最频繁，占总访问量的15%。

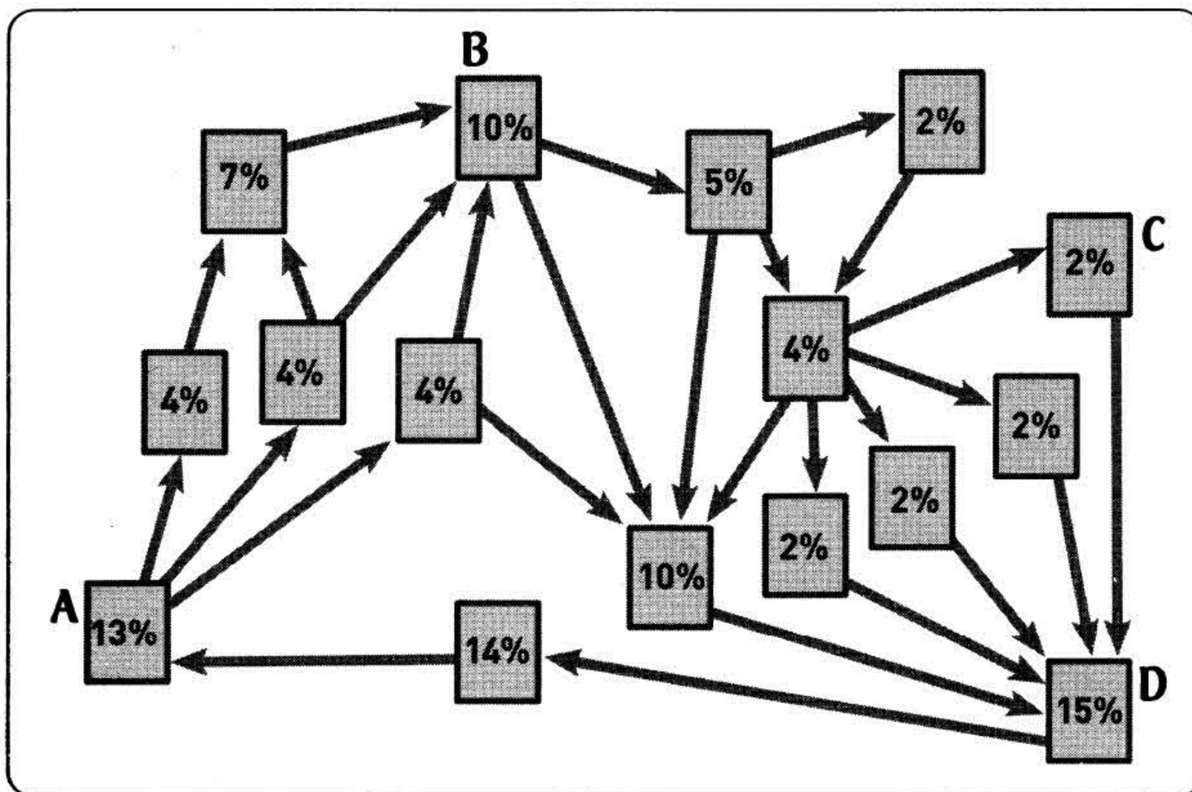
随机访问者模型和权重把戏之间有什么联系可以被我们用于网页排名呢？从随机访问者模拟中计算得出的百分比，正好就是我们在衡量一个网页的权重时所需要的。因此，让我们将网页的访问者权重值（surfer authority score）定义为一名随机访问者花在访问该网页的时间比例。值得注意的是，访问者权重值能和前两个对网页重要性进行排名的把戏配合良好。我们会逐一审视这些把戏。

首先，让我们来审视一下超链接把戏：超链接把戏的主要思想是，一个有许多链入链接的网页应该有高排名。这在随机访问者模型中也适用，因为一个有许多链入链接的网页被访问的概率较大。下图（底图）中的网页D就是个好例子：它有五个链入链接——比模拟中的其他网页都多——访问者权重值也最高（15%）。

其次，让我们来看看权重把戏。权重把戏的主要思想是，和来自低权重网页的链入链接相比，一个来自高权重网页的链入链接应该更能证明一个网页的排名。随机访问者模型也包含这一点。为什么？因为和一个来自不知名网页的链接相比，访问者更有可能继续点击一个来自知名网页的链入链接。要在我们的模拟中找这样一个例子，请比

较上面底图中的网页A和C：这两个网页都有一个链入链接，但网页A的访问者权重值要高得多（13% VS 2%），这主要取决于其链入链接的质量。






随机访问者模拟。顶图：1000次访问模拟中各网页的访问次数。
底图：100万次访问模拟中各网页的访问次数占比。

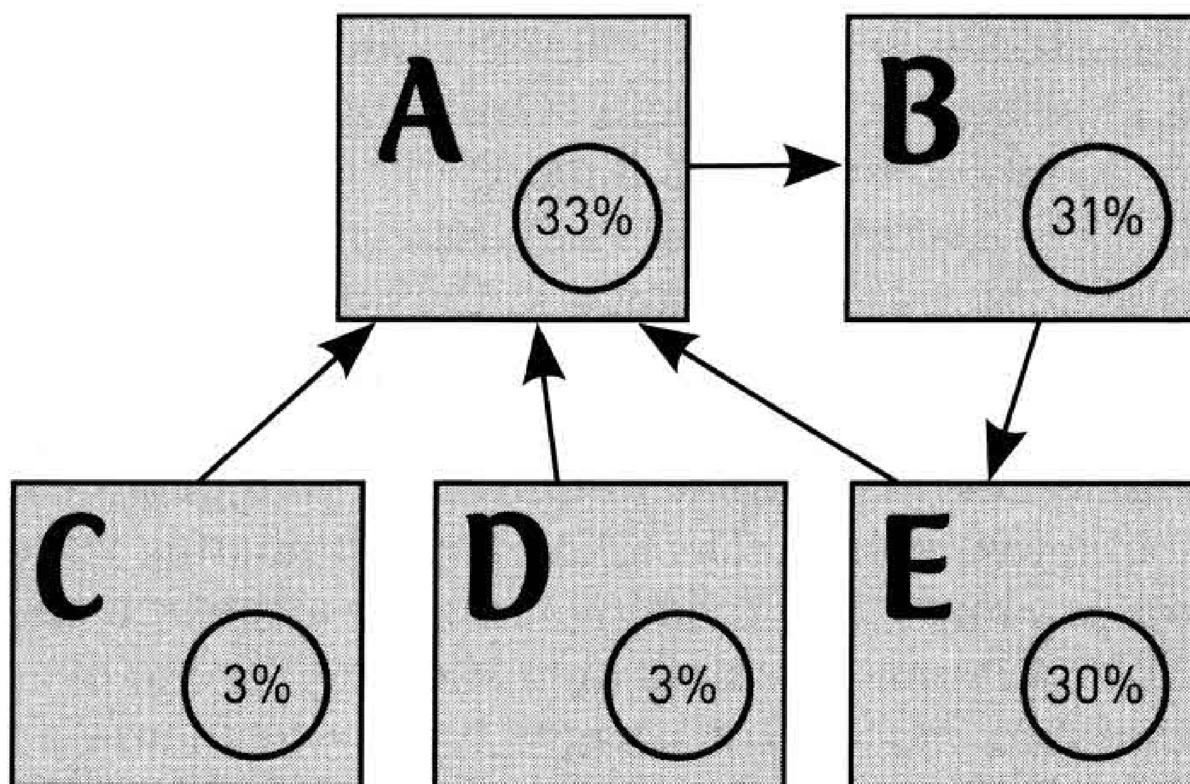
注意，随机访问者模型天生能同时和超链接把戏及权重把戏相配合。

前文炒蛋例中各网页的访问者权重值。伯特的菜谱网页和欧尼的菜谱网页都只有一个链入链接传入权重，但伯特的网页在网络搜索查询“scrambled eggs”（炒蛋）时排名会更高。

换句话说，每个网页链入链接的质量和数量都会被纳入考虑范围。网页B就展示了这些：网页B的访问者权重值相对较高（10%），得益于三个链入链接所在的网页拥有适中的访问者权重值，从4%到7%不等。

随机访问者把戏的美妙之处在于，和权重把戏不同，不管超链接有没有形成循环，随机访问者把戏都能完美地运作。回到早前的炒蛋

例子，我们能轻易地运行一次随机访问者模拟。在数百万次访问之后，我的模拟产生了如上图所示的访问者权重值。




早前超链接循环例子（此图）的访问者权重值。随机访问者把戏在计算合理的分值上没有问题，尽管存在一个循环（A-B-E-A)

请留意，和之前使用权重把戏进行的计算一样，伯特的网页访问者权重值要比欧尼的网页高很多（28%VS 1%）——尽管这两个网页都只有一个链入链接。因此，伯特的网页在网络搜索查询“scrambled eggs”（炒蛋）中排名更高。

现在让我们再转向前文中更困难的例子：对于最初的权重把戏而言，由于超链接循环的存在，这张图产生了一个不可解的问题。和前面一样，运行一次随机访问者的计算机模拟很容易，于是产生了如上图所示的访问者权重值。由这一模拟判定的访问者权重值给出了网页的最终排名，这些排名会被搜索引擎在返回结果时使用：网页A排名最高，之后是B和E，C和D的排名同列最后一名。

## 实际中的PageRank

谷歌的两位联合创始人于1998年在他们著名的会议论文《解析大规模超文本网络搜索引擎》中描述了随机访问者把戏。通过和其他许多技术结合，这一把戏的变体仍被主流搜索引擎所使用。不过，由于众多复杂因素，应用在现代搜索引擎中的实际技术和本章描述的随机访问者把戏略有不同。

其中一个复杂因素直击PageRank的核心：有时候，假设超链接传输的合法权威性有争议。我们先前已了解到，尽管超链接能代表批评而非推荐，但这在现实中并不是个很大的问题。另一个更加严重的问题是，人们可以滥用超链接把戏，人为地提高自己网页的排名。假设你运营着一个名为BooksBooksBooks.com的网站来售书（惊讶吧）。通过使用自动化技术，创建一大堆不同的网页——比如一万个——并让这些网页都链向BooksBooksBooks.com，做到这一切相对很容易。因此，如果搜索引擎和本章描述的一样来计算PageRank权重，BooksBooksBooks.com的权重值就能比其他书店高数千倍，进而有更高的排名和更多的销售额，而这都不是BooksBooksBooks.com应得的。

搜索引擎称这种滥用为网络垃圾（web spam）。（这一术语是和电子邮件垃圾<e-mail spam>类比得来的：电子邮件收件箱中无用的信息，类似于充斥在搜索结果中无用的网页。）对于所有搜索引擎而言，侦测并消除不同类型的网络垃圾是一直在进行的重要任务。比如，在2004年，微软一些研究人员发现，逾30万个网页都只有1 001个网页链向它们——这是件非常令人生疑的事情。通过手动检查这些网页，研究人员发现，这些链入超链接绝大多数都是网络垃圾。

因此，搜索引擎和网络垃圾制造者在进行一场军备竞赛。搜索引擎不断尝试完善算法，以便返回真实排名。在完善PageRank算法的驱动下，孕育了大量针对其他使用互联网超链接结构进行网页排名的算法的学术和行业研究。这类算法通常被称为基于链接的排名算法（link-based ranking algorithms）。

另一个复杂因素与PageRank计算的高效性有关。访问者权重值是通过运行随机模拟来计算的，但在整个互联网上运行这类模拟耗时太长，不能进行实际运用。因此，搜索引擎并非通过模拟随机访问者来计算PageRank值：它们使用能像随机访问者模拟一样给出相同答案的数学技巧，但计算成本要低很多。我们研究访问者模拟技术是因为它直观的吸引力，也因为它描述了搜索引擎计算什么，而非如何计算。

另外，值得一提的还有，商业搜索引擎中用来判定排名的算法，要比PageRank这类基于链接的排名算法多得多。即便是在他们于1998年发表的描述谷歌的原始论文中，谷歌的联合创始人也提到了多种对搜索结果排名有贡献的功能。正如你所想的，这项技术已经进步了：在写作本书时，谷歌官网上声明“有超过200个信号”被用于评估一个网页的重要性。

除了现代搜索引擎的众多复杂性之外，PageRank核心的优美思想——权威性网页通过超链接向其他网页传输权重——仍然有效。正是这一思想帮助谷歌击败了AltaVista，让谷歌从一家小型创业企业几年后成长为搜索之王。没有PageRank的核心思想，绝大多数搜索引擎查询都将被成千上万命中但不相关的网页海洋所淹没。PageRank的确是一块算法瑰宝，能让针毫不费力地冒到草垛的顶端。

# 第四章　公钥加密——用明信片传输秘密

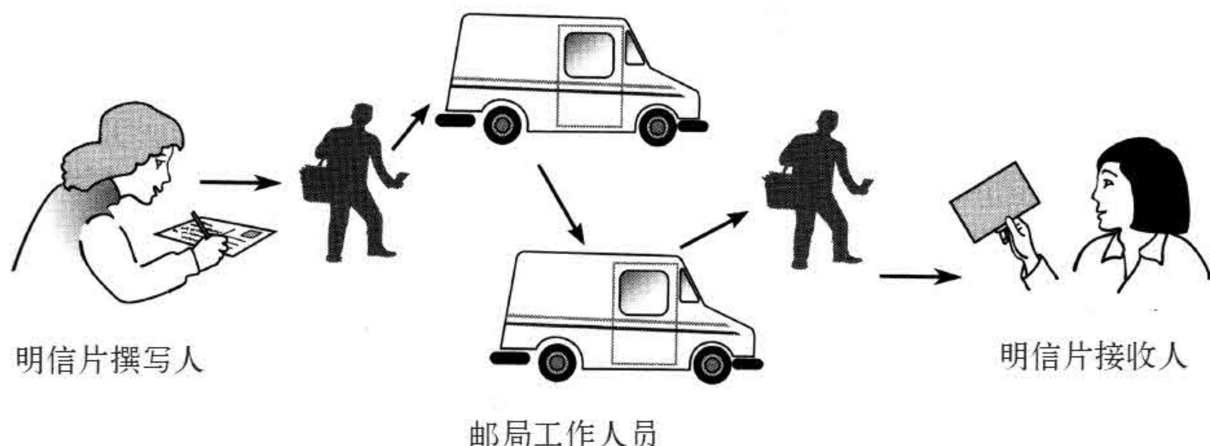
谁知道我这些最隐秘的事情？它们就藏在这个世界中。

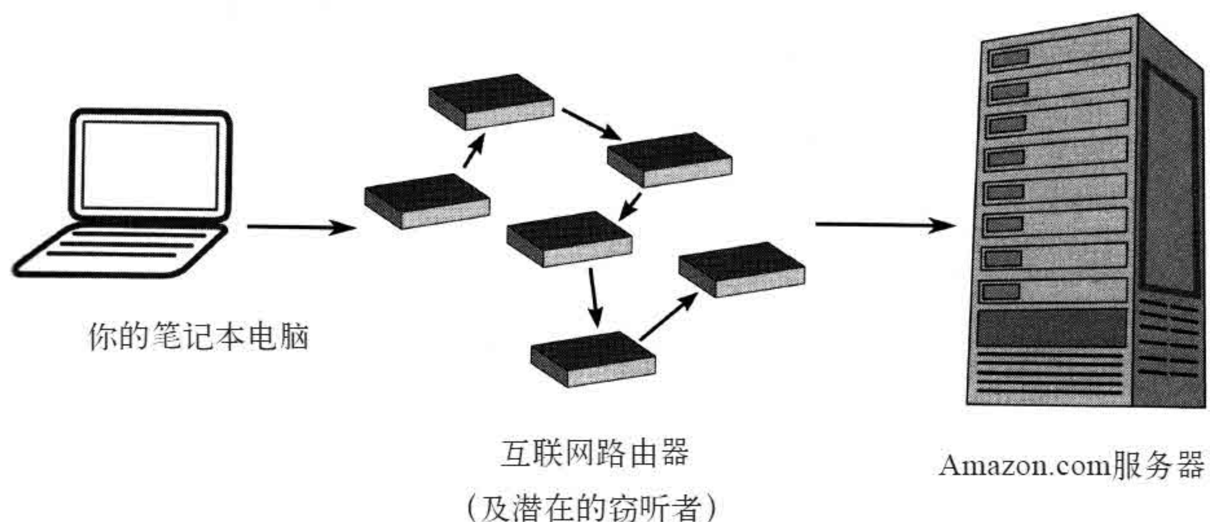
——鲍勃·迪伦，
《立约的女人》（Covenant Woman）

人们喜欢传谣，也喜欢了解秘密。而由于加密的目的就是传输秘密，所以我们都是天生的密码员。但人类进行秘密沟通要比计算机容易。如果你想向朋友透露一个秘密，你只需在朋友耳边低声说就行。计算机要做到这一点就不那么容易了。一台计算机没有办法“低声”向另一台计算机透露一张信用卡的卡号。如果计算机由互联网相连，它们控制不了信用卡卡号的流向，也无法控制会有哪些计算机知道卡号。在本章，我们会探究计算机如何应对这一问题——通过运用永远都称得上最精巧、最具影响力的计算机科学思想之一：公钥加密（public key cryptography）。

读到这里，你也许会想，为什么本章的标题中会提到“用明信片传输秘密”？下页图会给出答案：可以用通过明信片传输作为类比，以展示公钥加密的威力。在现实生活中，如果你想要发送一份机密文件给某人，你自然会在发送前将文件封存在一个安全的密封的信封内。这并不能保证机密性，但却是正确方向上的一个合理步骤。相反，如果在发送机密消息前，将机密信息写在明信片背面，很明显，这违背了机密性：任何一个接触过明信片的人（比如邮局工作人员）都只需查看明信片，就能读到这条消息。

这也正是计算机在互联网上尝试相互进行机密通信时面临的问题。因为互联网上的所有消息都会通过无数被称为路由器的计算机，消息的内容可以为任何访问路由器的人所见，而这也包括潜在的恶意窃听者。因此，每一块离开你计算机并进入互联网的数据，就好像写在明信片上！




明信片类比：很显然，通过邮政系统寄出一张明信片不能保证明信片内容的秘密性。基于同样的理由，如果没有恰当地加密，从你的笔记本电脑上将一张信用卡的卡号发送给Amazon.com,很容易就会被一名窃听者窥探到。

针对这个明信片问题，你也许已经想到了一个快速补救方案。在将消息写到明信片上之前，为什么我们不使用密码对每条消息进行加密呢？实际上，如果你已经知道接收明信片的人，这个方法还能奏效，因为在过去某个时刻，你们就使用哪种密码已经达成一致。真正

的问题出现在你给不认识的人寄明信片时。如果你在明信片上使用密码，邮局工作人员就不能读取你的消息，收信人当然也不能！公钥加密的真正威力在于，它允许你应用只有接收方才能解密的密码——除非你不能就使用哪种密码达成一致。

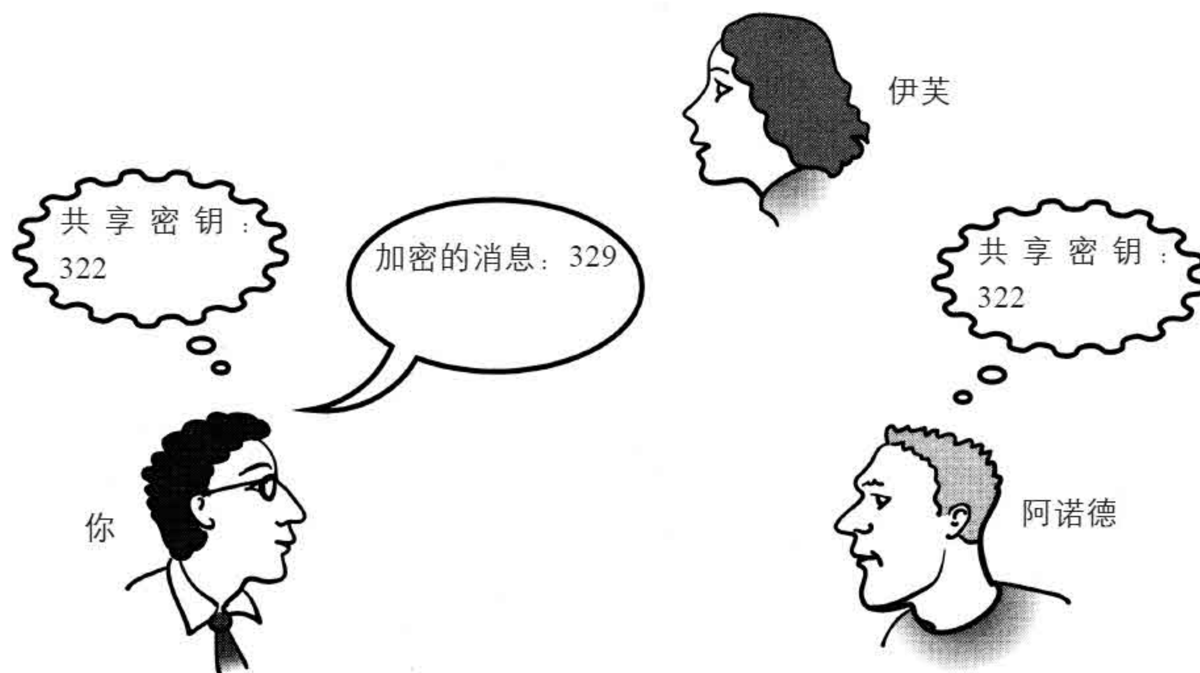
注意，计算机在和不“认识”的接收方通信时会面临相同的问题。比如，你第一次用信用卡在Amazon.com上购物时，你的计算机必须将你的信用卡卡号传输给亚马逊的服务器。但你的计算机之前从未和亚马逊的服务器联系过，因此这两台计算机在过去没有机会就密码达成一致。而它们之间尝试达成的任何协议都会被互联网上所有的路由器注意到。

让我们回到明信片的类比。的确，这种情况听起来像个悖论：接收方看到的信息和邮局工作人员看到的一样，但接收方却有某种手段能解码消息，而邮局工作人员却没有这种手段。公钥加密为这一悖论提供了一个解决方案。本章将解释其工作原理。

## 用共享密钥加密

让我们从一个非常简单的思想实验开始。我们将放弃明信片类比，用一些更简单的来代替：在一个房间内的口头交流。具体情况是，你和朋友阿诺德以及敌人伊芙待在一个房间里。你想要秘密地传输一条消息给阿诺德，但又不能让伊芙理解这条消息。这条消息可能是一张信用卡的卡号，但为方便起见，假设这是一个极其短的信用卡卡号，只是1到9之间的一个数字。而你只能通过大声说话和阿诺德联系，好让伊芙听到。不得偷偷摸摸地耍小把戏，比如小声说或传纸条。

具体来说，让我们假设你试图传输的信用卡卡号为7。有一种方法可以让你达成目标：首先，尝试想一些阿诺德知道而伊芙不知道的数字；比如，你和阿诺德成为朋友很久了，从孩提时就生活在同一条街上。事实上，假设你俩经常在你家前院玩耍，门牌号是愉快街322号。其次，假设伊芙并不是从小就认识你，特别是她不知道你和阿诺德过去经常玩耍的房屋地址。你就能对阿诺德说："嘿，阿诺德，记得我在愉快街的家的门牌号吗？我们过去一起玩耍的地方。如果你用门牌号，加上我现在想到的信用卡卡号中的一个数，你会得到329。"




相加把戏：通过将消息7和共享密钥322相加，消息7被加密了。阿诺德可以通过减去共享密钥提取消息7，但伊芙却不能。

现在，只要阿诺德能正确地记得房屋门牌号，他就可以通过将你告诉他的总数329和房屋门牌号相减，得到信用卡卡号。他用329减去322得7，这也是你试图向他传输的信用卡卡号。同时，伊芙却不知道信用卡卡号是多少，尽管她能听到你跟阿诺德说的每个词。上图显示了整个过程。

为什么这种方法能奏效？因为你和阿诺德有一样东西，也就是计算机科学家们所谓的共享密钥：数字322。因为你俩都知道这个数字，但伊芙却不知道，你可以使用这个共享密钥秘密地传输任何数字，只要和共享密钥相加，说出总数，让另一方减去共享密钥即可。听到总数对伊芙没有任何用处，因为她不知道要减去哪个数字。

不管你是否相信，如果理解了这个简单的“相加把戏”——将一个共享密钥和如一张信用卡的卡号这类私人消息相加——你就理解了互联网上绝大多数加密真正的运作原理！计算机不断地运用着这一把戏，但要真正实现保密性，还有一些细节需要注意。

首先，计算机使用的共享密钥要比房屋门牌号322长很多。如果密钥太短，任何窃听对话的人都可以试遍所有可能性。比如，假设我们使用相加把戏，用一个3位数房屋门牌号来加密一个真正的16位数信用卡卡号。注意，3位数房屋门牌号有999种可能，那么像伊芙这样窃听了对话的敌人可以列出一个包含所有999个可能数字的表，而其中肯定有一个数字是信用卡卡号。而计算机几乎不用费什么时间就能试遍999个信用卡卡号，因此为了共享密钥能起作用，我们需要用远长于3位数的数字来作为共享密钥。

事实上，当你听到某种加密是一个特定位数的数字，比如“128位加密”，这实际上说的是共享密钥的长度。共享密钥通常被称为“钥”，是因为它能用于解锁或“解密”一条消息。如果你解开了钥匙数位的30%，这也就告诉了你钥匙的大致数位。因为128的30%大约是38，我们就知道128位加密使用的钥匙是一个38位的数[注]。一个38位数要比10的36次方还要大，而又因为这需要任何现存的计算机花费数十亿年时间来试遍这么多可能性，一个38位数的共享密钥被认为非常安全。

这个简单相加把戏要在现实生活中奏效还要克服一个困难：加法得出的结果能用于统计分析，这意味着一些人能通过分析大量你的加

密消息来得到钥匙。相反，被称为“分块密码”（block cipher）的现代加密技术使用了相加把戏的变体。

首先，长消息被分解成固定大小（通常是10~15个字母）的小“块”。其次，和简单地将一块消息与钥匙相加不同的是，每个块都会根据一系列固定规则转换数次。这些规则类似于加法，但会让消息和钥匙更紧密地混合在一起。比如，规则可以是“将钥匙的前半部分和这块消息的后半部分相加，倒置结果，再将钥匙的第二部分和这块消息的前半部分相加”——不过在现实中，这些规则要更加复杂一些。现代分块密码基本上会进行10“轮”或更多类似操作，即操作列表会被反复应用。在转换的轮数足够多时，原始消息会真正地混合好，并能抵御统计攻击，但任何知道钥匙的人都能用相反的步骤运行所有操作，以获得最初的解密的消息。

在写作本书时，最流行的分块密码是高级加密标准（Advanced Encryption Standard），或称AES。AES能配合多种不同配置使用，但标准配置是使用16个字母的块，配备128位密钥，进行10轮混合操作。

## 公开建立一个共享密钥

到目前为止，一切情况良好。我们已经知道互联网上绝大多数加密技术的运作原理：将消息分成块，使用加法把戏变体加密每个块。但这是加密简单的地方。难点在于一开始要建立一个共享密钥。在上面的例子中，在你和阿诺德及伊芙待的房间里，其实我们做点了弊——我们利用了你和阿诺德从小玩到大的事实，因此阿诺德知道共享密钥（你家的房屋门牌号）而伊芙不可能知道。如果你、阿诺德和伊芙都是陌生人，我们怎么玩同样的游戏？你有没有办法和阿诺德建立一个伊芙不知道的共享密钥？（记住，不能作弊——你不能低声跟阿

诺德说任何事情或给阿诺德一张伊芙看不到的纸条。所有沟通都必须公开。）

乍一看，要做到这一点似乎不可能，但还是有个精巧的办法能解决这个问题。计算机科学家们称这一解决方案为迪菲—赫尔曼密钥交换（Diffie-Hellman key exchange），但我们把它称作颜料混合把戏（paint-mixing trick）。

## 颜料混合把戏

要理解这一把戏，我们先不管传输信用卡卡号的事，而是假设你想要分享的密钥是一种特殊颜色的颜料。（的确，这有点诡异，但我们很快会看到，用这种方法来思考这个问题非常有用。）现在假设你和阿诺德、伊芙待在一个房间内，每人都有大量不同的颜料桶。你们都拥有相同的颜色选择——有多种不同颜色，每个人都拥有多桶相同颜色的颜料。这样就不存在用完颜料的问题了。每一桶颜料都清楚地标示了其颜色，因此在具体指导其他人混合不同颜色的颜料上就很容易了：你只要说些如“将一桶‘天蓝色’颜料和六桶‘淡黄褐色’颜料以及五桶‘碧绿色’颜料混合在一起”的话即可。但在每一块已知的色板（shade）上都有上百或上千种颜色，因此，不可能只通过看颜色就知道其中混合了哪些具体的颜色。而且也不可能通过试错发现混合颜色中加入了哪些具体的颜色，因为可以尝试的颜色太多了。

现在要改变一下游戏规则。你们三人各占据房屋的一角，每个角落都出于隐私考虑加以屏障，你可以在其中存放颜料，在其他人看不到的情况下混合颜料。但沟通规则和之前一样：在你、阿诺德和伊芙之间的任何沟通都必须公开。你不能邀请阿诺德进入你的私人混合区域！另一条规则规定了你分享颜料混合配方的方式。你可以给屋内其他人一批颜料，但只能把颜料放到房间中央的地板上，等其他人来捡

起它。这也意味着，你永远也不能确定谁会捡起你放的颜料。最好的办法是，为每个人提供足够多的颜料，然后在房间中央留下数批分开的颜料。这样，任何想要你颜料的人都可以拿取。这条规则其实只是所有沟通都必须公开的补充：如果你给了阿诺德某种混合颜料，却没有给伊芙，你就和阿诺德进行了某种“私密”沟通，这违反了规则。

记住，这个颜料混合游戏旨在解释如何建立一个共享密钥。在这时，你也许在想混合颜料和加密有什么关系，请不要着急。你将了解到一个令人惊奇的把戏，计算机使用这一把戏在像互联网这样的公共场合建立共享密钥！

首先，我们要知道这个游戏的目标。目标是让你和阿诺德都能制作相同的混合颜料，而不能让伊芙知道如何生产。如果你达成了这一目标，我们就能说你和阿诺德建立了一种“共享的秘密混合颜料”。你可以随心所欲地进行尽可能多的公开对话，你也可以携带颜料桶多次往返于房间中央及你的私人混合区域之间。

现在我们要开始探访公钥加密背后精巧思想的旅程了。我们的颜料混合把戏分为四步：

第一步：你和阿诺德各自选择一种“私人颜色”。

你的私人颜色与你最终将制造的共享秘密混合颜料不同，但它将是共享秘密混合颜料的成分之一。你可以选择任何一种颜色作为私人颜料，但你必须记住这种颜色。很显然，你的私人颜色几乎肯定会和阿诺德的私人颜色不同，因为可供选择的颜色太多了。假设你选了淡紫色作为私人颜色，阿诺德选了深红色作为私人颜色。

第二步：选择一种新的不同的颜色成分并公开宣布，我们称这种颜色为“公开颜色”。

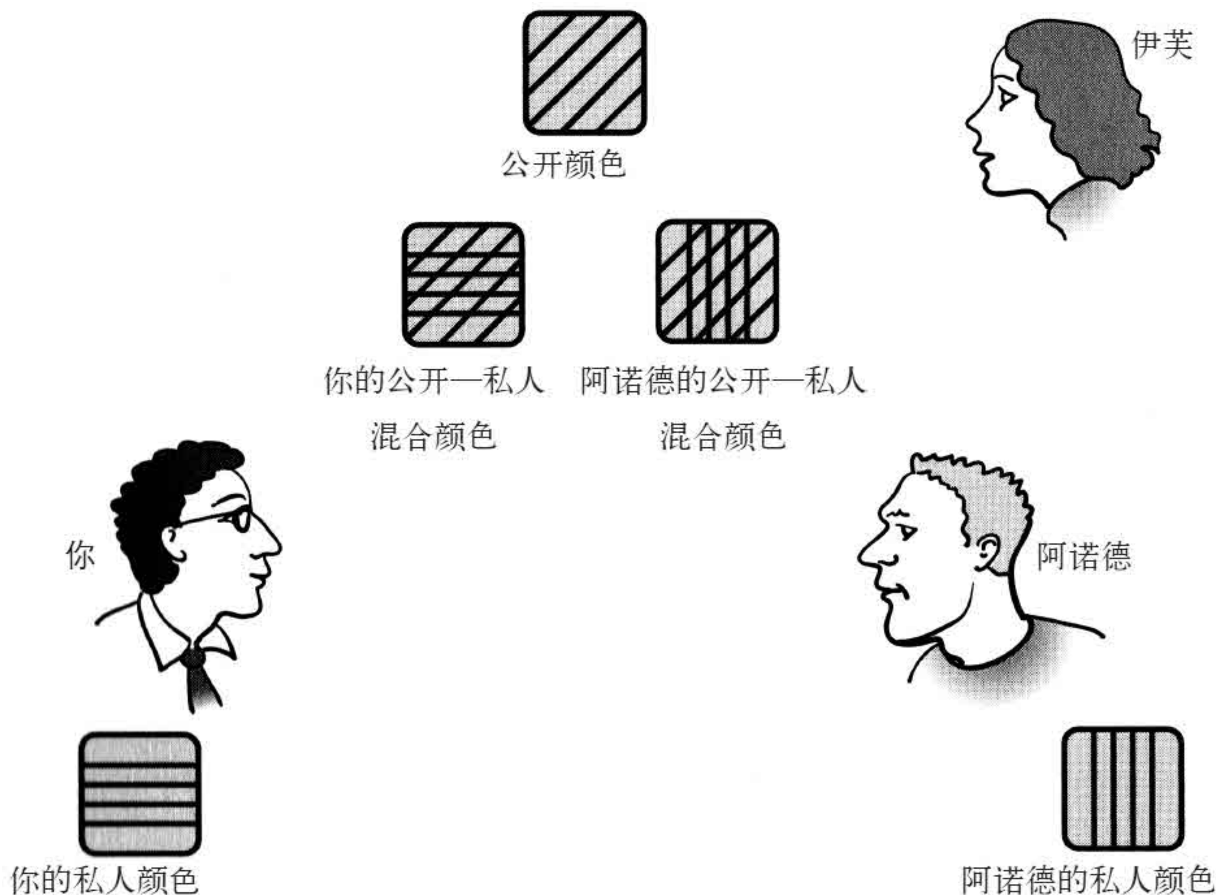
和之前一样，你可以选择任何颜色。假设你宣布的公开颜色是雏菊黄。注意只能有一种公开颜色（而不是你和阿诺德各选一种公开颜色），伊芙自然也知道这种公开颜色，因为你公开宣布了。

第三步：你和阿诺德各用一桶公开颜色和一桶私人颜色制造一种混合颜色。这就是你的“公开—私人混合颜色”。

很显然，阿诺德的公开—私人混合颜色会和你的不同，因为他的私人颜色和你的不同。如果继续使用上面的例子，你的公开—私人混合颜色会包含一桶淡紫色和一桶雏菊黄，而阿诺德公开—私人混合颜色会包含一桶深红色和一桶雏菊黄。

到这时，你和阿诺德会想给彼此公开—私人混合颜色的样品，但记住，直接给房间中一个人混合颜料是违反规则的。给其他人一种混合颜色的唯一方法是制作数批该种颜料，并把它们放到房间中央，以便任何想要的人拿取。这也正是你和阿诺德所做的：你们俩都制作数批公开—私人混合颜色，并把它们放到房间中央。如果伊芙想要的话，她可以偷走一到两批，但我们很快就会了解到，这些颜料对伊芙没有任何用处。下页的图显示了颜料混合把戏第三步之后的情况。

现在到达关键点了。如果在这时仔细想一会儿，你可能会知道最后的把戏会让你和阿诺德制造出同一种共享的秘密混合颜色，而不让伊芙知道秘密。答案如下：

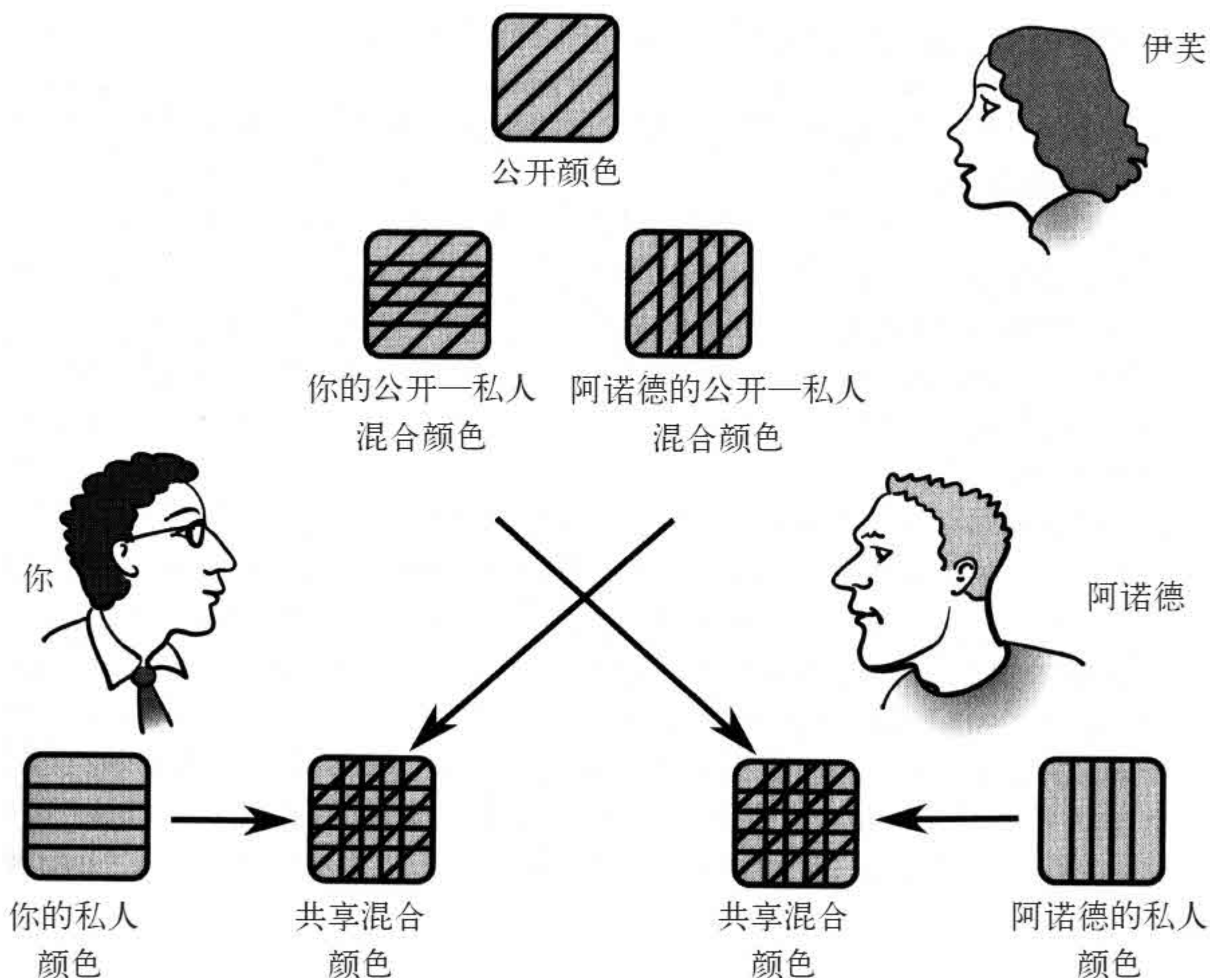


颜料混合把戏第三步：任何想要的人都可以拿取公开—私人混合颜色。

> 第四步：你选取一批阿诺德的公开—私人混合颜色，拿回自己的角落。现在加入一桶私人颜色。同时，阿诺德选取一批你的公开—私人混合颜色，拿回他的角落，在那里他再加入一桶他的私人颜色。

神奇吧，你俩制作了同样的混合颜色！让我们来检验一下：你将自己的私人颜色（淡紫色）加入阿诺德的公开—私人混合颜色（深红色和雏菊黄），得到的最终混合颜色是：一桶淡紫色、一桶深红色和一桶雏菊黄。阿诺德最终得到的混合颜色呢？他将自己的私人颜色（深红色）加入你的公开—私人混合颜色（淡紫色和雏菊黄），得到的最终混合颜色是：一桶深红色、一桶淡紫色和一桶雏菊黄。这正和

你得到的最终混合颜色一样，也恰好是一种共享秘密混合颜色。下页的图显示了颜料混合把戏最后一步的情形。




颜料混合把戏第四步：只有你和阿诺德能制作这种共享秘密颜色，如箭头所示组合混合颜色。

那伊芙呢？为什么她不能制作一份这种共享秘密混合颜色呢？原因是她不知道你或阿诺德的私人颜色，而她至少需要其中一种来制作共享秘密混合颜色。你和阿诺德打败了她，因为你从未在房间中央公开过自己的私人颜色。相反，你在公开前将自己的私人颜色和公开颜色混合在一起，而伊芙没有办法“分开”公开—私人混合颜色，也就不能获得其中一种私人颜色的纯正样本。

因此，伊芙只能获取两种公开—私人混合颜色。如果她将一桶你的公开—私人混合颜色和一桶阿诺德的公开—私人混合颜色混合在一起，结果是包含了一桶深红色、一桶淡紫色和两桶雏菊黄。换句话说，和共享秘密混合颜色相比，伊芙的混合颜色多了一份雏菊黄。她的混合颜色太黄了，而因为没有办法“分开”颜料，她不能移除多余的黄色。你也许会想，伊芙可以通过加入更多深红色和淡紫色来达到目的，但要记住，伊芙不知道你的私人颜色，因为她不会知道这些颜色还需要加入的颜色。她只能加入深红色配雏菊黄的组合或淡紫色配雏菊黄的组合，而这会导致她的混合颜色太黄。

## 用数字进行颜料混合把戏

如果你理解了颜料混合把戏，你就会理解计算机在互联网上建立共享密钥的核心机制。当然，它们并不真的使用颜料。计算机使用数字，而要混合数字，它们会运用数学。计算机实际使用的数学并不会太复杂，但也复杂到让人第一眼看上去就会感到困惑。因此，作为理解共享密钥如何在互联网上建立的下一步，我们将用到一些“伪装”数学。真正的要点在于，要将颜料混合把戏转化成数字，我们需要一种单向操作（one-way action）：可以做一些事情，但不能取消做过的事。颜料混合把戏中的单向操作是“混合颜料”。将一些颜料混合起来形成一种新颜色很容易，但要“分开”它们并获得原来的颜料则不可能。这也是为什么颜料混合是单向操作的原因。

我们在前面提到过，将会用到一些伪装数学。下面就是我们要伪装的：将两个数相乘是一项单向操作。我敢肯定，你已经意识到了，这绝对是个借口。乘法的反面是除法，只需要用除法就能惊奇地取消乘法。比如，如果我们从数字5开始，然后乘以7，得到35。要取消这个乘法很容易，只要从35开始，除以7就可以了。这会得到我们一开始时的数字5。

但除此以外，我们将坚持使用这一伪装，在你、阿诺德和伊芙之间玩另一个游戏。这一次，我们将假设你们都知道如何将数相乘，但你们都不知道如何用一个数除以另一个数。目标和前面的游戏几乎一样：你和阿诺德试图建立一个共享密钥，但这次共享密钥将是一个数，而非一种颜料。此前的沟通规则也适用：所有联系必须公开进行，这样伊芙就能听到你和阿诺德之间的任何对话。

好，现在我们要做的事情就是将颜料混合把戏转换成数字：

> 第一步：和选择一种“私人颜色”不同的是，你和阿诺德选择一个“私人数字”。

假设你选择了4，阿诺德选择了6。现在回想颜料混合把戏剩余的步骤：宣布公开颜色，制作公开—私人混合颜色，公开地将你的公开—私人混合颜色与阿诺德的公开—私人混合颜色交换，最后将你的私人颜色加入阿诺德的公开—私人混合颜色，以获得共享秘密颜色。要理解如何将这些步骤转换成数字，并用乘法作为单向操作取代颜料混合也不至于太难。在继续往下读之前，花几分钟看看你能否自行理解这个例子。

遵照这个解决方案并不太难；你和阿诺德都选择了自己的私人数字（4和6），下一步就是：

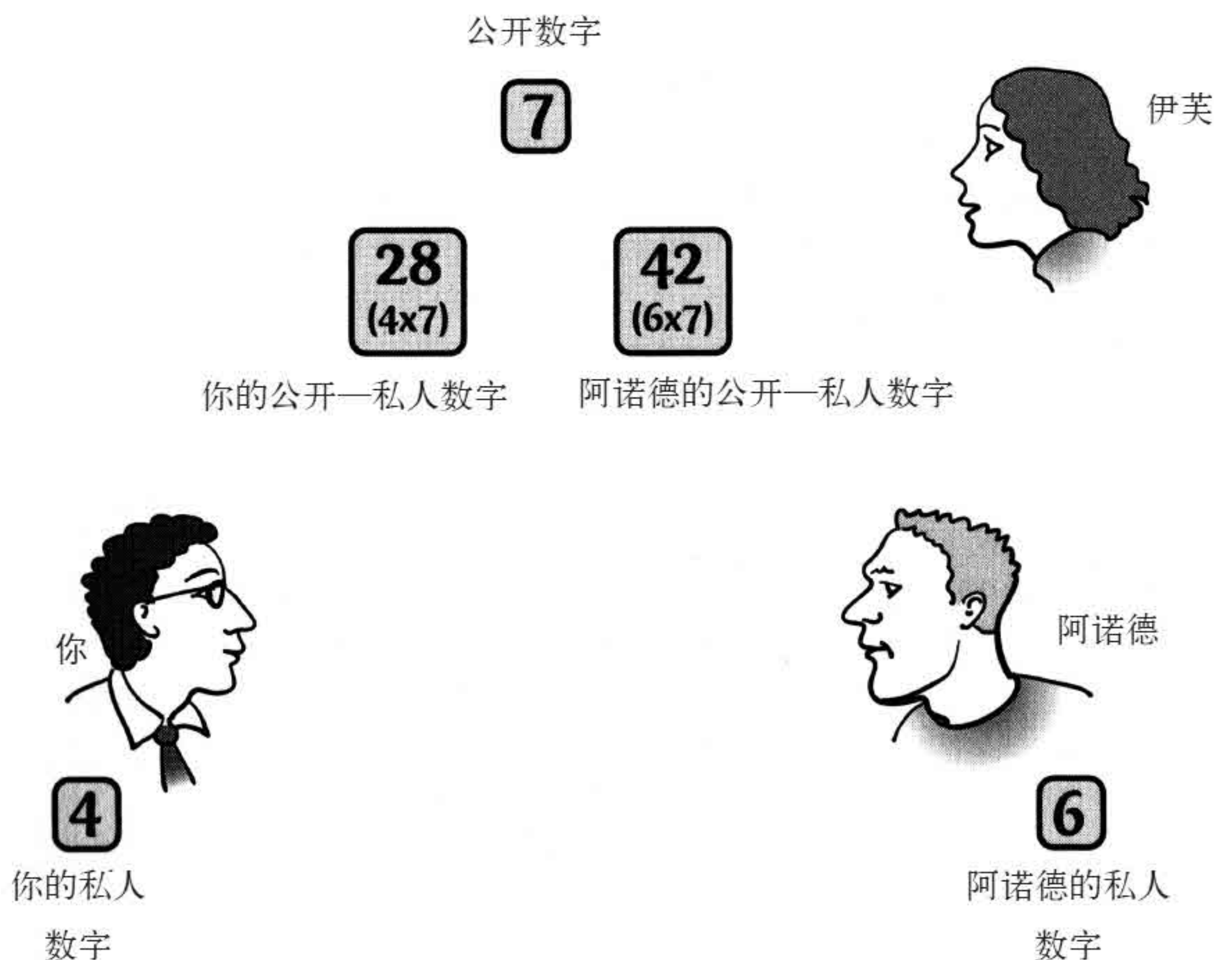
> 第二步：你们其中一个人宣布“公开数字”（取代颜料混合把戏中的公开颜色）。

假设你选择了7作为公开数字。

颜料混合把戏的下一步是制作一份公开—私人混合颜色。但我们已经决定，用将数字相乘来代替混合颜料。因此，你要做的就是：

第三步：将你的私人数字（4）和公开数字（7）相乘，得到“公开—私人数字”28。

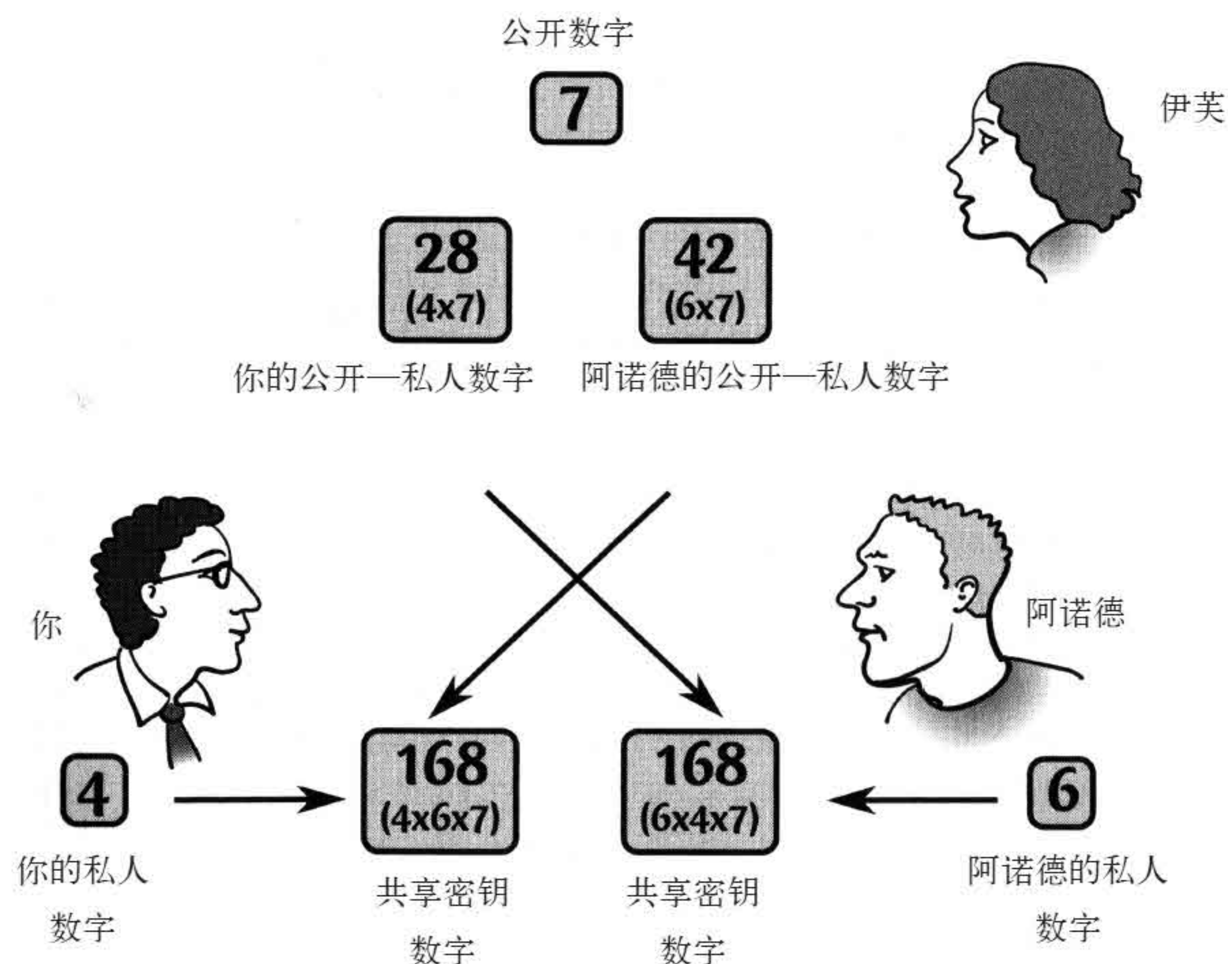
你可以公开地宣布，这样阿诺德和伊芙就都知道了你的公开—私人数字是28（无须再携带颜料桶了）。阿诺德对自己的私人数字做了同样的事：他将自己的私人数字和公开数字相乘，并宣布他的公开—私人数字是42。下面的图显示了整个流程到达这一点的情形。



还记得颜料混合把戏的最后一步吗？你拿走阿诺德的公开—私人混合颜色，加入一桶你的私人颜色，以制作共享秘密颜色。这里发生的事情也一模一样，用乘法代替颜料混合：

> 第四步：你把阿诺德的公开—私人数字（42）和你的私人数字（4）相乘，结果是共享秘密数字168。

同时，阿诺德用你的公开—私人数字28和他的私人数字6相乘，并令人惊讶地得到了共享秘密数字168。最终结果显示在下图中。




数字混合把戏第四步：只有你和阿诺德能得到共享密钥数字，通过将途中箭头所示的数字相乘。

事实上，当你用正确的方法思考时，这一点都不令人惊讶。阿诺德和你都制作出了相同的共享秘密颜色的原因是，你将相同的三种原始颜色混合在了一起，但却使用不同的顺序：你俩各自保留了一种私人颜色，把它和由其他两种颜料组成的公开混合颜料组合在一起。同样的事也发生在数字上。你俩通过将同样的三个数4、6和7 相乘，得到了相同的共享密钥。（你可以自己查证，$4 \times 6 \times 7 = 168$。）但你之所以能达到这一目标，是通过用私人数字4和由6乘7得出的公开混合数字“混合”（相乘）得出的，而阿诺德也宣布了这个混合数字。另一方面，阿诺德通过用私人数字6和由4乘7得出的公开混合数字“混合”（相乘）得出共享密钥，而你也宣布了这个混合数字。

就和我们在颜料混合把戏中做的一样，让我们来验证伊芙不能得到共享密钥。每个公开—私人数字的值在宣布时都能被伊芙听到。她听到你说28，阿诺德说42。她还知道公开数字是7。因此，如果伊芙知道如何做除法，她就能马上知道你所有的密钥，只需要观察到$28 \div 7 = 4$和$42 \div 7 = 6$即可。而她可以继续通过计算$4 \times 6 \times 7 = 168$算出共享密钥。不过，幸运的是，我们在游戏中使用的是伪装数学：我们假设乘法是一种单向操作，因此伊芙不知道如何相除。于是她被数字28、42和7难住了。她可以将其中一些数相乘，但结果和共享密钥无关。比如，如果她将$28 \times 42 = 1\ 176$，那就大错特错了。就和颜料混合游戏中她的结果太黄了一样，在这里，她的结果中有太多7。共享密钥中只有一个7，因为$168 = 4 \times 6 \times 7$。但伊芙想要破解密钥的要素中有两个7，因为$1\ 176 = 4 \times 6 \times 7 \times 7$。而且她还没有办法去掉多余的7，因为她不知道如何做除法。

## 现实生活中的颜料混合把戏

我们现在已经讲完了计算机在互联网上建立共享密钥所需的所有基本概念。使用数字的颜料混合机制的唯一缺陷是，它使用的是“伪装

数学”，在其中我们假设没有任何一方能做除法。要完成整个流程，我们需要一个在现实生活中容易做到（比如颜料混合），但在实际中又不可能取消（比如分离颜料）的数学操作。当计算机在现实生活中进行这些操作时，混合操作就是离散指数（discrete exponentiation），而分离操作则被称为离散对数（discrete logarithm）。由于还没有一种方法能让计算机高效地计算离散对数，离散指数就成了我们寻找的那类单向操作。要恰当地解释离散指数，我们需要两个简单的数学概念。我们还需要写几个公式。如果你不喜欢公式，请直接忽略余下的部分——你几乎了解了和这个主题有关的所有东西。另外，如果你真的想知道计算机如何做这件神奇的事，接着往下读。

我们需要的第一个重要的数学概念被称为钟算（clock arithmetic）。这倒是我们都熟悉的东西：钟上有12个数字，每次当时针路过12时，都会从1开始重新计数。一个从10点钟开始、持续4小时的活动会在2点钟结束，也就是说，在这个12小时钟系统中，10+4 = 2。在数学中，钟算也是这样运行的，除了两个细节：（1）钟的大小可以是任何数（而非一个普通的钟上熟悉的12个数字），（2）数字从0而不是从1开始计数。

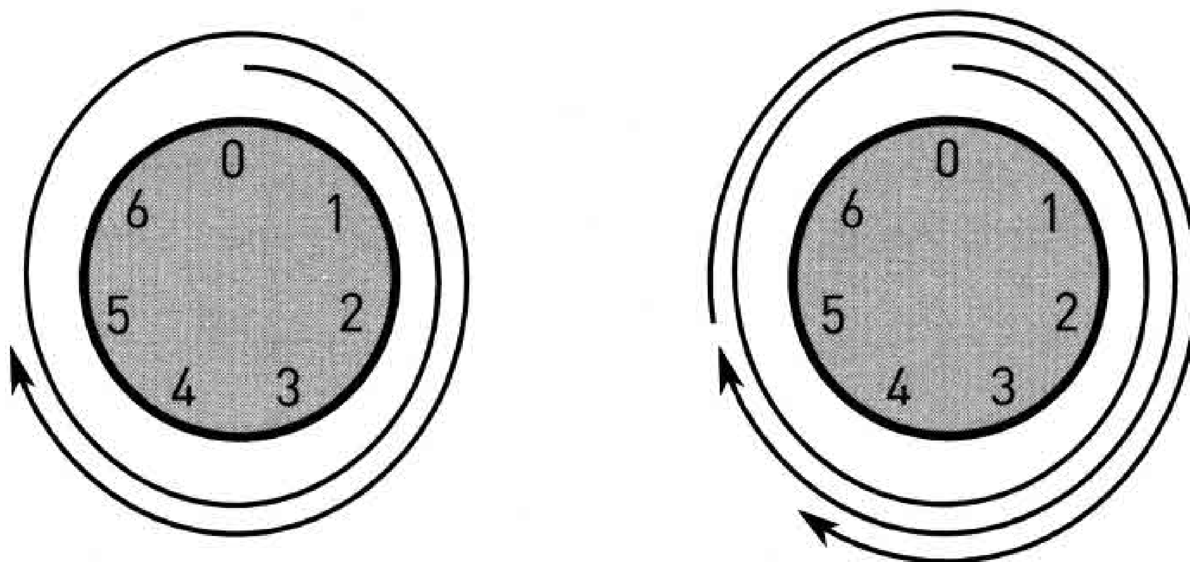
下面图中的例子用的是大小为7的钟。注意，钟上的数字是0、1、2、3、4、5和6。用钟大小为7的钟做钟算，只要像平常一样将数字相加相乘即可，不过不管结果如何，你只要取除以7所得的余数即可。计算12 + 6，我们首先像平常一样相加，得到18。然后我们注意到18中有两个7（即14），余4。因此最终答案是：

$$12 + 6 = 4 \text{（钟大小为7）}$$

在下面的例子中，我们将钟大小调为11。（稍后将讨论到，现实应用中的钟大小要大很多。我们使用小的钟大小是为了让解释尽可能简单。）在将结果除以11后取余并不太难，因为11的倍数都是像66和88这样重复的数。下面是几个用钟大小为11进行计算的例子：

$$7 + 9 + 8 = 24 = 2 \quad (\text{钟大小为}11)$$

$$8 \times 7 = 56 = 1 \quad (\text{钟大小为}11)$$




我们需要的第二个数学概念是幂函数（power notation）。这个概念一点也不花哨：就是写下许多相同数字相乘的快捷方法。和写下$6 \times 6 \times 6 \times 6$不同的是，你可以写成$6^4$，这就是指6乘以本身4次。你可以将幂函数和钟算结合起来。比如：

$$3^4 = 3 \times 3 \times 3 \times 3 = 81 = 4 \quad (\text{钟大小为}11)$$

$$7^2 = 7 \times 7 = 49 = 5 \quad (\text{钟大小为}11)$$

下页的表格显示了钟大小为11时数字2、3和6的前10个幂。这在我们将要研究的例子中非常有用。在继续深入前，请你确保自己认可这张表格的生成方式。让我们来看看最后一栏。这一栏的第一项是6，也就是$6^1$。下一项代表$6^2$，即36，但由于我们使用的钟大小为11，36比33大3，因此这一项是3。要计算这一栏的第三项，你也许会认为我们要计算$6^3 = 6 \times 6 \times 6$，但有个更简单的方法。我们已经用自己感兴趣的钟大小计算了$6^2$，结果是3。要得到$6^3$，我们只需要将先前的结果乘

6。也就是$3 \times 6 = 18 = 7$（钟大小为11）。下一项是$7 \times 6 = 42 = 9$（钟大小为11），依此类推，直到该栏最后一项。

最后，我们终于要准备建立一个计算机在现实生活中使用的共享密钥了。和之前一样，你和阿诺德将尝试分享一个密钥，而伊芙窃听并试图算出密钥。

第一步：你和阿诺德各自单独选择一个私人数字。

| $n$ | $2^n$ | $3^n$ | $6^n$ |
|---|---|---|---|
| 1 | 2 | 3 | 6 |
| 2 | 4 | 9 | 3 |
| 3 | 8 | 5 | 7 |
| 4 | 5 | 4 | 9 |
| 5 | 10 | 1 | 10 |
| 6 | 9 | 3 | 5 |
| 7 | 7 | 9 | 8 |
| 8 | 3 | 5 | 4 |
| 9 | 6 | 4 | 2 |
| 10 | 1 | 1 | 1 |

这张表显示了钟大小为11时数字2、3和6的前10个幂。正如上文所解释的，每个项都能通过一些非常简单的算术从上一项中算出。

为保证数学尽可能简单，我们将在这个例子中使用非常小的数字。因此，假设你选择8作为私人数字，而阿诺德选择9。这两个数字——8和9——都不是共享密钥，而是像你在颜料混合把戏中选择的私人颜色，这些数字将被用作“混合”一个共享密钥的成分。

第二步：你和阿诺德公开就两个公开数字达成一致：钟大小（本例中使用11）和另一个被称为基数的数字（选2为基数）。

这些公开数字——11和2——像公开颜色一样，你和阿诺德在颜料混合把戏一开始就对此达成了一致。注意，颜料混合类比与本例有点不同：在颜料混合中，我们只需一种公开颜色，而在这个例子中，我们需要两个公开数字。

第三步：通过使用幂符号和钟算，你和阿诺德各自将自己的私人数字和公开数字相混，分别得到一个公开—私人数字（public–private number，PPN）。

具体来说，混合是按照公式来完成的：

$$PPN = base^{私人数字} \quad （钟大小）$$

这个公式用文字写出来可能会显得有点诡异，但在实际中却很简单。在我们的例子中，我们能通过参考上一页的表格来得出答案：

$$你的PPN = 2^8 = 3 \quad （钟大小为11）$$

$$阿诺德的PPN = 2^9 = 6 \quad （钟大小为11）$$

在这一步之后，你可以在下页的图中查看这一步。这些公开—私人数字正相当于你在颜料混合把戏第三步中制作的“公开—私人混合颜色”。在那一步中，你将一桶公开颜色和一部分你的私人颜色相混合，制作你的公开—私人混合颜色。在这里，你使用幂符号和钟算将自己的私人数字和公开数字相混合。

第四步：你和阿诺德各自单独获得对方的公开—私人数字，再和自己的私人数字相混合。

这可以按照下面的公式完成：

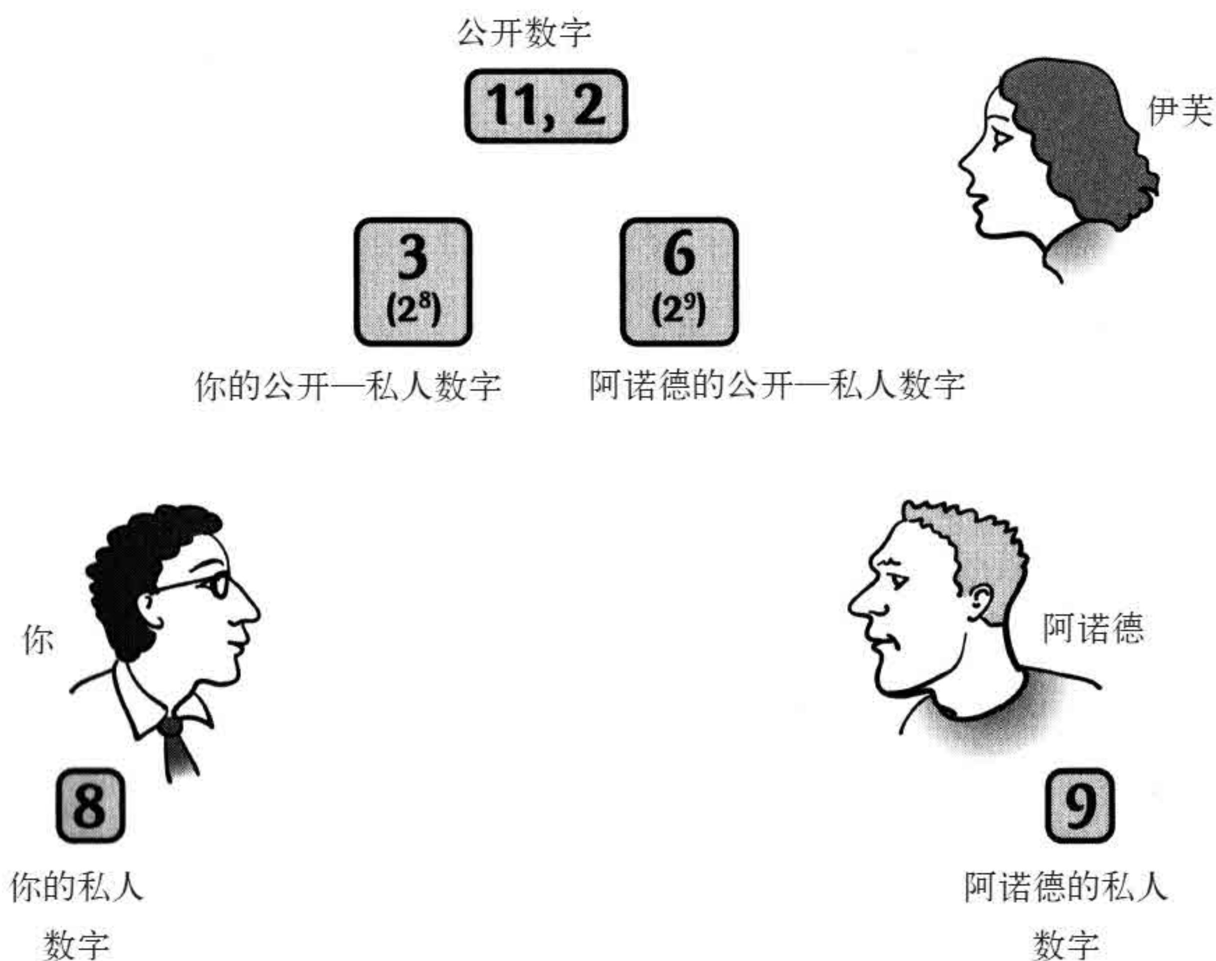
$$共享密钥=其他人的PPN^{私人数字}\quad（钟大小）$$

和前面的公式一样，用文字写出来会显得有点诡异，但通过参考前一页的表格，很容易就能得到答案：

$$你的共享密钥= 6^{8}= 4\quad（钟大小为11）$$

$$阿诺德的共享密钥= 3^{9}= 4\quad（钟大小为11）$$

最终解决方案显示在下一节的图中。



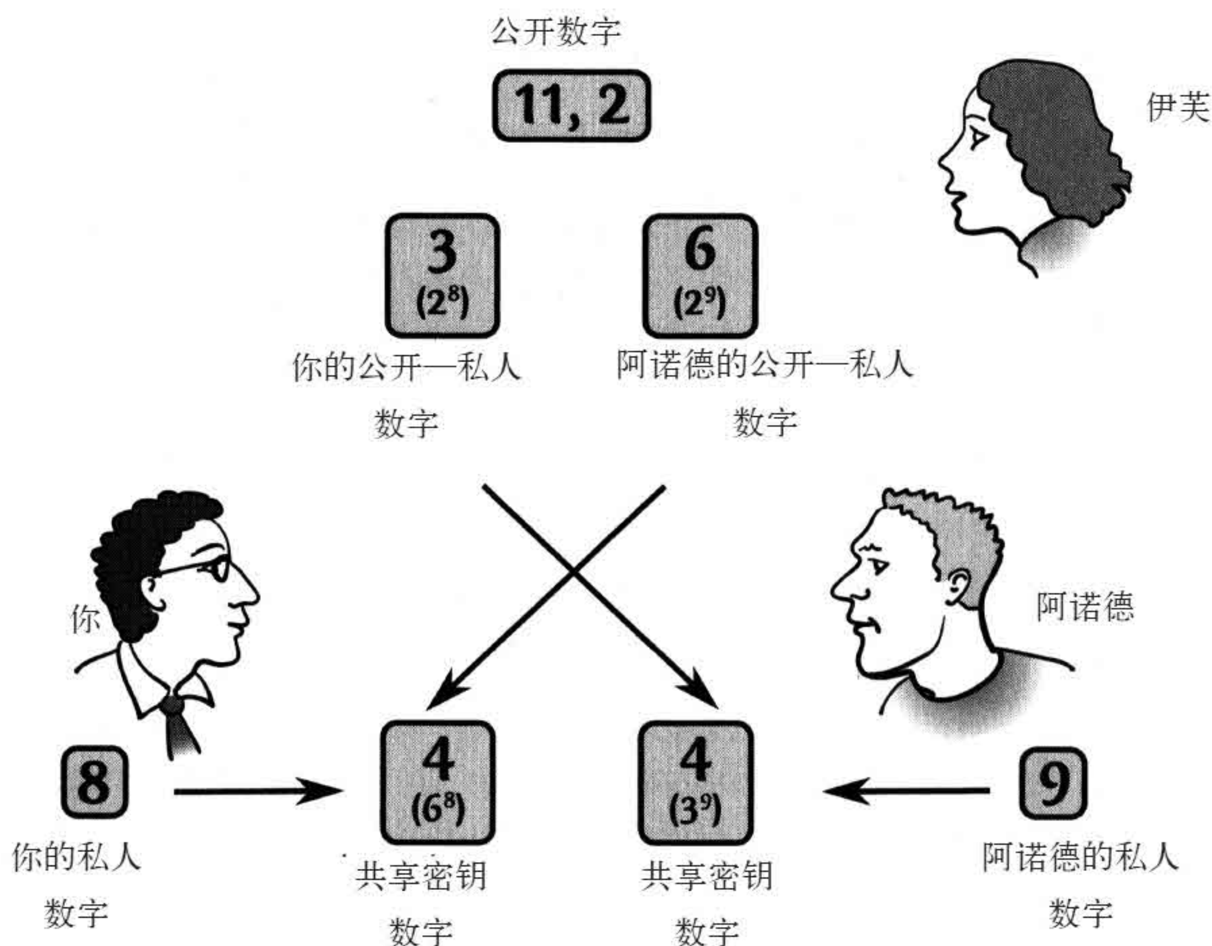

现实生活数字混合第三步：任何想要公开—私人数字（3和6）的人——使用幂和钟算计算得出——都可以获取到。3下面的“$2^8$”提醒了我们3是如何计算的，但$3=2^8$使用的钟大小为11这个事实却并未公开。类似的，6下面的“$2^9$”也仍然保持私密。

自然，你的共享密钥和阿诺德的共享密钥相同（这个例子中是4）。取得这个结果依靠了例子中的一些复杂算术，但基本概念和前面一样：尽管你按照不同的顺序混合了各种成分，但你和阿诺德都使用了相同的成分，因此也得到了相同的共享密钥。

和这个把戏的早前版本一样，伊芙被排除在外。她知道这两个公开数字（2和11），她也知道这两个公开—私人数字（3和6）。但她不能使用任何知识来计算共享密钥数字，因为她不能获取你和阿诺德持有的秘密成分（私人数字）。

## 实际中的公钥加密

颜料混合把戏的最终版本——运用幂和钟算混合数字——就是计算机在互联网上建立共享密钥的实际方法之一。本章描述的方法被称为迪菲—赫尔曼密钥交换机制。这一机制以怀特菲德·迪菲（Whitfield Diffie）和马丁·赫尔曼（Martin Hellman）的名字命名，他俩于1976年首次发表了这一算法。每次你访问一个安全网站（以“https:”而非“http:”开头），你的计算机和其通信的网站服务器之间都会建立一个共享密钥，使用的方法是迪菲—赫尔曼机制或工作原理类似的替代方法之一。而一旦建立了共享密钥，这两台电脑就能使用之前描述的相加算法的变体对所有通信进行加密。



现实生活数字混合第四步：只有你和阿诺德能得到共享密钥数字，通过使用幂和钟界计算将如箭头所示的元素组合起来。

还有很重要的一点值得注意，当迪菲—赫尔曼机制在现实中运用时，实际涉及的数字要远比我们在例子中使用的数字大。我们使用的钟大小很小（11），因此计算起来就很简单。但如果你选择的公开钟大小很小，那么可能的私人数字也会很小（因为你只能使用比钟大小小的私人数字）。而这也意味着有人可以使用计算机试出所有可能的私人数字，直到找到一个数字生成你的公开—私人数字。在上面的例子中，只有11个可能的私人数字，因此这个系统非常轻易就能被破解。相反，迪菲—赫尔曼机制在现实中运用时通常会使用有几百个数位长的钟大小，这让可能的私人数字多得令人不可想象（比一万亿的一万亿次方还多得多）。即便如此，也需要小心选取公开数字，以确保它们具有正确的数学性质——如果你感兴趣的话，请看下页的文字框。

迪菲—赫尔曼公开数字最重要的属性是，钟大小必须是一个素数——只有1和其自身两个除数。另一个有趣的要求是，基数必须是钟大小的本原根（primitive root）。这也意味着基数的幂最终将循环遍钟上每一个可能的值。再看看前文提到的表，你会注意到2和6都是11的本原根，但3却不是——3的幂循环到的值有3、9、5、4和1，却没有2、6、7、8和10。

在为迪菲—赫尔曼机制选择一个钟大小和基数时，钟大小和基数必须满足特定的数学性质。

本章描述的迪菲—赫尔曼方法只是通过（电子）明信片通信的多种绝妙技巧之一。计算机科学家们称迪菲—赫尔曼方法为密钥交换算法（key exchange algorithm）。其他公钥算法的运作方式不同，能让你使用接收方宣布的公开信息，直接向潜在的接收方加密一条消息。相反，密钥交换算法能让你使用来自接收方的公开信息，建立一个共

享密钥，但加密过程通过加法把戏来完成。对于互联网上绝大部分通信而言，后面的选项——我们在本章中了解到的这种方法——要更容易让人接受，因为它需要的计算能力要少很多。

但也有一些程序需要用到完整的公钥加密。这类程序中最有趣的可能要算数字签名了，我们将在第九章谈到它。在你阅读第九章时会发现，完整版公钥加密的思想类似于我们已经了解到的：机密信息和公开信息用一种在数学上不可逆的方式“混合”在一起，就像混合在一起的颜料一样，再也分不开。最著名的公钥加密系统是RSA，这个名字由首次发表该系统的三位发明者姓的首字母组合而成：罗纳德·李维斯特（Ronald Rivest）、阿迪·沙米尔（Adi Shamir）和雷奥纳德·阿德尔曼（Leonard M.Adlemen）。第九章用RSA作为解释数字签名如何运行的主要例子。

在这些早期公钥算法发明的背后，都有一个迷人而复杂的故事。迪菲和赫尔曼的确是发表迪菲—赫尔曼机制的第一批人，时间是1976年。李维斯特、沙米尔和阿德尔曼也的确是发表RSA的第一批人，时间是1978年。但这并不是故事的全部！人们后来发现，英国政府在数年前就已经知道类似系统。不幸的是，那些发明迪菲—赫尔曼机制和RSA的先驱们是英国政府通信实验室GCHQ的数学家。他们工作的结果被记录在内部机密文件中，直到1997年才解密。

RSA、迪菲—赫尔曼机制和其他公钥加密系统不仅仅是绝妙的思想。它们还发展成了商业技术和互联网标准，对商业和个人有着极其重要的意义。没有公钥加密，我们每天使用的绝大部分在线交易都不可能安全地完成。RSA的发明者在20世纪70年代为自己的系统申请了专利，而他们的专利直到2000年年末才失效。在专利失效的那天晚上，美国旧金山市的美国音乐大剧院举行了一次庆祝晚会——也许是为庆祝公钥加密将永久为人们服务这一事实。

---

1. 对于了解计算机数字系统的人而言，我在这里说的是小数位数，而非二进制数。对于了解对数的人而言，将二进制数转化为小数位数的转换比例30% 来自于$log10^2 \approx 0.3$。

# 第五章　纠错码——自纠正的错误

告诉一个人他犯了错是一回事，让他掌握真理则是另一码事。

——约翰·洛克（John Locke），《人类理智论》（Essay Concerning Human Understanding）

如今，我们已习惯于在有需要时随时使用计算机。20世纪40年代在贝尔电话公司实验室工作的研究员理查德·汉明（Richard Hamming）就没这么幸运了：其他部门使用着他所需要的公司计算机，只有周末才轮到他用。因此，你可以想象，由于计算机在读取自身数据时出错从而导致频繁崩溃后，他有多么沮丧。汉明谈到这个问题时是这样说的：

> 我要轮两周才能进去一次，却发现我所有的东西都被清空了，什么都没得到。我真的很生气，因为我想要这些答案，而且还浪费两个周末的时间。于是我说："去他的，如果计算机能侦测到错误，为什么它不能对错误定位并进行纠正？"

要说明发明纠错码的必要性，还有另外几起很鲜明的例子。汉明很快就创造了第一批纠错码：一种近乎神奇的能侦测并纠正计算机数据中错误的算法。没有纠错码，我们的计算机和通信系统会比现在慢很多，功能上弱许多，可靠性也会差很多。

# 错误侦测及纠正的需求

计算机有三项基本工作。最重要的工作是执行计算，即给予计算机一些输入数据，计算机必须用某种方法转化数据，并得出一个有用的答案。但如果没有计算机执行的另外两项非常重要的工作：存储数据和传输数据，计算答案的能力基本上也没用。（计算机通常在内存和磁盘驱动器上存储数据，基本上是通过互联网传输数据。）要深入理解这一点，想象一台既不能存储也不能传输信

息的计算机。这样的计算机自然不会有什么用。的确，你可以做一些复杂计算（比如，准备一份复杂的财务报表，详细说明公司预算），但你不能将结果发送给同事，甚至不能保存结果以便返回继续操作。因此，传输和存储数据对现代计算机是真正的至关重要。

但传输和存储数据要面临一个巨大挑战：数据必须丝毫无差——因为在许多情况下，哪怕一丁点错误也会让数据变得毫无用处。作为人，我们也熟悉在不出错的情况下存储和传输信息的需求。比如，如果你写下某人的电话号码，按正确的顺序无误地记录下每位数至关重要。哪怕其中有一位数出错，那个电话号码对你或其他人都毫无用处。在一些情况中，数据出错的后果要比毫无用处更糟。比如，存储一个计算机软件的文件中有一个错误，这会导致程序崩溃或让程序做一些原本不应该做的事。（程序甚至可能会删除一些重要文件，或在你保存工作前崩溃。）而在一些计算机化的金融记录中出错，可能会导致实际的金钱损失。（假如你以为自己在买每股5.34美元的股票，而这只股票的实际价格是每股8.34美元。）

不过，人们需要存储的无误信息相对较少，而且在知道一些信息——如银行账号、密码、电子邮件地址等——很重要的情况下，通过仔细检查避免错误也不是太难。而另一方面，计算机要无误地存储和

传输的信息量绝对是海量。为了让你对信息量级有概念，想想下列情形。假设你有一个存储容量是100 GB的计算设备。（在写作本书时，这是低价笔记本电脑的标准容量。）这100 GB相当于1 500万页文本。因此，即便计算机存储系统每100万页犯一个错误，在设备容量填满时（平均）也会有15个错误。这一情况也适用于传输数据：如果你下载一个20兆的软件，你的计算机每接收100万个字符就会出错一次，下载的软件中也会有逾20个错误——在你不曾预料到的情况下，每一个错误都有可能导致成本巨大的崩溃。

这个故事的教训是，对于计算机而言，精确度达到99.999 9%也还是不够好。计算机必须能在存储和传输数十亿块信息的情况下，不犯任何一个错误。但和其他设备一样，计算机也必须处理通信问题。电话就是个好例子：显然电话不能完美地传输信息，因为电话对话经常要遭遇失真、静电噪声或其他类型的噪声。但电话并不是遭遇这类情况的唯一设备：电线受所有种类的波动影响；无线通信无时无刻不受干扰；硬盘、CD（激光唱盘）和DVD（数字视频光盘）等物理媒体会由于灰尘或其他物理干扰的影响，被划伤、受损或不能读取。面临如此显然的通信错误，我们怎么能希望实现数十亿分之一的错误率呢？本章将揭示让这一奇迹发生的绝妙计算机科学背后的思想。结果证明，如果你运用正确的把戏，即便是极端不可靠的通信频道也可以以极低的错误率传输数据。而且这个错误率是如此之低，以至于在实际当中，错误基本上完全被消除了。

## 重复把戏

要通过一个不可靠的频道进行可靠的通信，其中最根本的把戏是我们都熟悉的：要确保一些信息正确地传输，你只需重复几次该信息。如果某人在电话连接很糟糕的情况下，念给你听一个电话号码或

银行账号，你极有可能会要求对方至少重复一次，以便确认号码无误。

计算机也能做同样的事情。假设银行的一台计算机试图通过互联网把你的账户余额传给你。你的账户余额是5 213.75美元，但不幸的是，网络不稳定，每一个数字都有20%的概率变成其他东西。因此，当你的账户余额第一次传输过来时，显示的可能是5 293.75美元。很显然，你没办法知道余额是否正确。所有的数字都有可能正确，但其中一个或以上的数字可能出错，而你没有办法分辨。但通过运用重复把戏（the reprtition trick），你能很好地推测出真正的余额。假设你请求传出余额5次，并收到了以下反馈：

传输1：＄5 2 9 3 · 7 5

传输2：＄5 2 1 3 · 7 5

传输3：＄5 2 1 3 · 1 1

传输4：＄5 4 4 3 · 7 5

传输5：＄7 2 1 8 · 7 5

注意，其中一些传输不止一位数出错，也有一次传输（传输2）没有出现错误。关键点在于，你没办法知道哪儿有错，因此也没办法将传输2挑选为正确的传输。相反，你可以做的事情就是单独检查每个数字，寻找同一数字的所有传输，然后选出出现最频繁的那个值。下面列出了所有传输项，最末尾是出现频率最高的数字：

传输1：＄5 2 9 3 · 7 5

传输2：＄5 2 1 3 · 7 5

传输3：＄5 2 1 3 · 1 1

传输4：＄5 4 4 3 · 7 5

传输5：＄7 2 1 8 · 7 5

出现频率最高的数字：＄5 2 1 3 · 7 5

要把这个概念完全弄清楚，让我们来看些例子。检查传输中的第一位数，在传输1~4中，第一位数都是5，而传输5的第一位数是7。换句话说，第一位数在4次传输中是“5”，而只在一次传输中是“7”。因此，尽管你不能完全肯定，但你银行余额第一位数的值最有可能是5。转到第二位数，我们看到2出现了4次，而4出现了1次，因此2最有可能是第二位数。第三位数有点有趣，因为有三种可能性：1出现了3次，9出现了1次，4出现了1次。但同样的原则也适用，1是最有可能为真的值。通过对所有数字使用这种方法，你可以得到完整银行余额的最终推测：5213.75美元，而这也正是正确的值。

这种方法很简单。我们已经解决这个问题了吗？在某种程度上，是的。但你也许会对两件事感到不满。首先，这个信道的错误率只有20%，而在一些情况中，计算机需要在错误率远高于20%的信道中通信。其次，也许要更严肃些，上面例子的最终答案恰好是正确的，但不能保证答案会永远都正确：它只是一个推测，基于我们的看法——认为它才是最有可能为真的银行余额。幸运的是，这两件事都很容易处理：我们只需增加重新传输的次数，直到可靠性高到让我们满意为止。

比如，假设最后一个例子中的错误率是50%而不是20%。你可以要求银行传输1 000次余额，而不是5次。让我们集中关注第一位数，因为其他数的工作原理都一样。由于错误率是50%，大约有一半的数会正确地传输为5，但另一半会变成其他随机值。因此5会出现约500

次，其他每个数（0~4和6~9）都会出现50次左右。数学家们能计算出某个数出现的次数比5多的概率：即便使用这个方法每秒传输一个新的银行余额，我们也得等上数万亿年才能猜错银行余额。这个故事的重点在于，通过重复一条不可靠的消息足够多次，你可以让消息的可靠性高到让你满意为止。（在这些例子中，我们假设错误随机发生。相反，如果一个恶意实体故意干扰传输，并选择制造哪些错误，重复把戏都要不安全得多。后面介绍的一些代码在对抗这类恶意攻击时都表现良好。）

因此，通过使用重复把戏，不可靠通信的问题能够被解决，错误率基本上能被消灭。不幸的是，重复把戏对于现代计算机系统来说还不够好。当传输像银行余额这样的小块数据时，重新传输1 000次耗费并不多，但在下载一个大型软件（假设有200兆）时，显然传输1 000份该软件完全不现实。很明显，计算机需要使用一些比重复把戏更成熟的方法。

## 冗余把戏

即便计算机不使用上面描述到的重复把戏，我们也在本章一开始介绍它，以便让我们能了解实际生活当中可靠通信最基本的原则。这条基本原则是，你不能只发送原始消息：你要发送一些多余的东西以增加可靠性。在重复把戏的例子中，你发送的额外东西就是更多份原始消息。但实际情况是，要提高可靠性，你还有其他许多多余的东西可以发送。计算机科学家们称这些多余的东西为“冗余”。有时候，冗余被附加在原始消息上。在研究下一个把戏（校验和把戏）时，我们会看到这种“附加”技术。但首先，我们要研究另一种添加冗余的方法。这种方法实际上是把原始消息转换成一条更长的冗余消息——原始消息会被删除，取而代之的是一条更长的不同消息。当收到更长的

消息时，你就能将其转换回原始消息，即便这条冗余消息在糟糕的信道中传输时被破坏了。我们将这种方法简单地称为冗余把戏。

举个例子更容易说清。之前我们尝试将你的银行余额5 213.75美元通过一条不可靠的信道传输，这条信道有20%的概率随机替换数字。和尝试只传输“＄5 213.75”不同的是，让我们把这个数字转换成一条包含相同信息的更长的（因此也是“冗余的”）消息。在这个例子中，我们用英语单词简单地把余额拼出来：

five two one three point seven five

我们再假设，由于信道糟糕，这条消息中约20%的字符会变成其他随机字符。这条消息最终可能变成：

fiqe kwo one thrxp point sivpn fivq

尽管读起来有点讨厌，但我认为，任何知道英语的人都能猜出，这条被破坏的消息代表真正的银行余额5 213.75美元，这点你应该会赞同。

关键在于，因为我们使用了一条冗余消息，对消息中的任何单个变化进行可靠侦测及纠正变得可行。如果我告诉你，字母“fiqe”代表英语中的一个数字，且只有一个字母被替换了，你绝对能肯定原始消息中的单词是“five”，因为除此以外再也没有英语数字能通过替换“fiqe”中一个字母获得了。与此形成鲜明对比的是，如果我告诉你，数字“367”代表了一个数，但其中一个数字被替换了，你没办法知道原始数字是多少，因为这条消息中没有冗余。

尽管我们还没弄清冗余究竟是如何工作的，但却已经知道它和让消息变长有关，消息的每一部分都应该符合某种已知模式。通过这种方法，任何变化都能首先被识别（因为并不符合已知模式），然后被

纠正（通过改变错误使其符合模式）。计算机科学家称这些已知模式为“代码字”（code words）。在上面的例子中，代码字就是用英语写的数字，如“one”、“two”、“three”等。

现在是时候解释冗余把戏（the redundany trick）究竟是如何运作的了。消息由“符号”——计算机科学家是这么称呼的——组成。在我们这个简单例子中，符号是数字0~9（我们会忽略美元符号以及小数点，让事情变得更简单）。每个符号都被指定了一个代码字。在这个例子中，符号1被指定的代码字是“one”，2被指定的代码字是“two”，依此类推。

要传输一条信息，你首先要找出每个符号，并将符号转译成对应的代码字。其次，你将转换的消息通过不可靠信道发送。当消息被接收到时，你查看消息的每个部分，检查其是否为有效的代码字。如果它是有效的（如“five”），你只需将其转换为相应的符号（如5）即可。如果其不是有效的代码字（如“fiqe”），你要找出其和哪个代码字最匹配（这个例子中就是“five”），并将那个无效代码字转换成相应的符号（也就是5）。使用这些代码的例子如下图所示：

**编码**

| | | |
|---|---|---|
| 1 | —→ | one |
| 2 | —→ | two |
| 3 | —→ | three |
| 4 | —→ | four |
| 5 | —→ | five |

**解码**

| | | |
|---|---|---|
| five | —→ | 5（完全匹配） |
| fiqe | —→ | 5（最接近的匹配） |

| twe | —→ | 2（最接近的匹配） |
|---|---|---|

使用英文字表示数字的代码

这就是全部过程。实际上，计算机在存储和传输信息时会一直用到这个冗余把戏。和我们在例子中使用的英语相比，数学家们已经找到了更漂亮的代码字，但可靠计算机通信的工作原理仍然相同。下页的图就列举了一个真实例子。这个例子被计算机科学家们称为（7，4）汉明代码（Hamming code），这是理查德·汉明于1947年在贝尔实验室发明的代码之一，为了处理之前说过的周末计算机崩溃。（由于贝尔实验室的要求，汉明申请了这些代码的专利，直到3年后的1950年才发表。）和我们在前面用到的代码相比，汉明代码最明显的区别是所有事情都通过0和1完成。因为计算机存储和传输的每一块数据都会被转化为0和1的字符串，现实生活中使用的所有代码也限用这两个数字。

**编码**

| | | |
|---|---|---|
| 0000 | —→ | 0000000 |
| 0001 | —→ | 0001011 |
| 0010 | —→ | 0010111 |
| 0011 | —→ | 0011100 |
| 0100 | —→ | 0100110 |

**解码**

| | | |
|---|---|---|
| 0010111 | —→ | 0010（完全匹配） |
| 0010110 | —→ | 0010（最接近的匹配） |
| 1011100 | —→ | 0011（最接近的匹配） |

计算机使用的真实代码。计算机科学家们称这一代码为（7,4）汉明代码。注意，“编码”框中只列出了16个可能的四位数输入中的5个。其余的输入也都有相应的代码字，但它们在这里被省略了。

除此以外，所有事情都和前面一模一样。在编码时，每一组4位数字都加入了冗余，产生了一个7位数的代码字。在解码时，你首先要为接收到的7位数寻找完全匹配，如果寻找完全匹配失败，就选择最接近的匹配。你也许会担心，现在我们在和0及1打交道，也许相近的匹配不只一个，最后可能会选择错误的解码。不过，这种特殊代码的设计非常精巧，7位数代码字中的任何错误都能得到确定无疑的纠正。在设计带有这种属性的代码背后是一些美丽的数学，但在这里，我们不会深究其细节。

值得强调的是，为什么冗余把戏在实际应用中要比重复把戏更受欢迎。主要原因是这两个把戏的相对成本。计算机科学家们使用“杂项”（overhead）衡量纠错系统的成本。杂项就是为确保消息被正确接收而发送的多余信息。重复把戏的杂项数量巨大，因为你必须发送数份完整消息。冗余把戏的杂项取决于你使用的代码字的具体类型。上面的例子使用英语作为代码字，冗余消息有35个字母长，而原始消息只含有6个数字，因此冗余把戏这一特殊应用的杂项也很大。但数学家们已经找到了冗余度低很多的代码字，而且其在侦测错误的概率上也效率惊人。这些代码字的低杂项也是计算机使用冗余把戏——而非重复把戏——的原因。

到目前为止，我们进行的讨论都使用利用代码传输信息的例子，但我们讨论过的所有事情都能很好地在存储信息的任务中应用。CD、DVD和计算机硬盘都极度依赖纠错码，以实现我们在现实中观察到的超级可靠性。

## 校验和把戏

目前为止，我们研究了同时侦测和纠正数据中错误的方法。重复把戏和冗余把戏都属于此类方法。但还有另外一种可能的方法能解决

整个问题：我们可以先不管纠错，而是将精力集中在侦测错误上。（17世纪的哲学家约翰·洛克对错误侦测和错误纠正之间的区别有清楚地认识——正如你在本章开篇引言中所看到的。）对于许多软件而言，只侦测到一个错误就足够了，因为如果你侦测到了一个错误，请求再发送一份数据即可。而且你可以一直请求新拷贝，直到得到完全无误的拷贝。这是最经常使用的策略。比如，几乎所有互联网连接都使用这一技术。我们称这一方法为“校验和把戏”（The checksum trick），这么命名的原因很快就会明朗。

要理解校验和把戏，假设我们所有消息都只有数字组成会更方便些。这是一个非常真实的假设，因为计算机用数字存储所有信息，只有在向人展示信息时，才把数字转译成文本或图像。不过，在本章所有的例子中，任何对消息符号的特殊选择都不影响本章描述的技术，理解这点很重要。有时候使用数字符号（数字0~9）更方便，而有时候使用字母符号（字母a~z）要更加方便。不过，不管是使用哪种符号，我们都能就这些符号之间的一些转译达成一致。比如，从字母转译为数字符号的一个明显例子就是a→01、b→02、……、z→26。因此，我们是在探究一项传输数字消息还是字母消息的技术就变得无关紧要了；通过首先对符号进行简单的固定的转译，这项技术稍后将被应用在任何种类的消息上。

到这里，我们必须了解校验和究竟是什么。校验和的种类有很多，但目前我们将着重于最简单的那种校验和——我们称之为“简单校验和”（simple checksum）。计算一条数字消息的简单校验和的确很容易：你只需将消息中的所有数字相加，只保留结果的最后一位数，剩下的数字就是你的简单校验和。比如：假设消息是：

4 6 7 5 6

那么所有数字之和为4+6+7+5+6 = 28，但我们只保留最后一位数，因此这条消息的简单校验和是8。

但如何使用校验和呢？很简单：你只需在发送原始消息前，将原始消息的校验和附加到消息末尾即可。在别人接收到消息后，他们能再计算校验和，并和你发送的校验和比较，看是否正确。换句话说，他们“check”（校验）消息的“sum”（和）——这就是术语“checksum”（校验和）的由来。继续说上面的例子。消息“46756”的简单校验和是8，因此我们这样来传输消息及其校验和：

4 6 7 5 6 8

现在，接收消息的人必须知道你使用的是校验和把戏。假设他们确实知道，他们就能立即认出最后一位数8不属于原始消息，然后把最后一位数放一边，计算其他数的和。如果在传输这条消息时没有出现错误，他们会计算4+6+7+5+6 = 28，保留最后一位数（也就是8），在将最后一位数和之前放到一边的校验和比较，看是否相等（的确相等），因此得出总结，消息正确地进行了传输。另一方面，如果传输消息时出错了，这时会发生什么？假设7随机变为3。那么你会收到如下消息：

4 6 3 5 6 8

你将8放到一边以备后续比较，并计算校验和为4+6+3+5+6 = 24，只保留最后一位数（4）。4和之前放到一边的8并不相等，因此你可以肯定消息在传输途中遭到了破坏。这时，你请求重新传输消息，直到接收到新拷贝，然后再次计算并比较校验和。你可以一直这么做，直到得到一条校验和正确的消息为止。

所有这一切似乎太过美好，不可能成真。还记得吗？一个纠错系统的“杂项”就是在发送消息本身以外要发送的额外信息量。好吧，我们似乎得到了终极低杂项系统，因为不管消息有多长，我们只需多添加一位数（校验和）就能侦测错误！

啊，结果证明简单校验和系统太过美好，不能成真。简单校验和的问题是：上面描述到的简单校验和最多只能在消息中侦测出一错误。如果有两个或更多错误，简单校验和或许能侦测到它们，但也有可能侦测不到。让我们来看看这个问题的一些例子：

| | | 校验和 |
|---|---|---|
| 原始消息 | 4 6 7 5 6 | 8 |
| 有一处错误的消息 | 1 6 7 5 6 | 5 |
| 有两处错误的消息 | 1 5 7 5 6 | 4 |
| 有两处（不同）错误的消息 | 2 8 7 5 6 | 8 |

原始消息（46756）和前面一样，其校验和也没变（8）。下一行的消息有一个错误（第一个数是1而不是4），其校验和为5。事实上，你很有可能说服自己，更改任何单个数字都会导致与8不同的校验和，因此你保证能侦测到消息中的任何单个错误。要证明这一点永远为真并不难：如果只有一个错误，简单校验和绝对能保证侦测到它。

再下一行，我们看到一条带有两处错误的消息：前两个数都被替换了。这条消息的校验和恰好是4。而由于4和原始校验和8不同，收到这条消息的人能侦测到出现了一个错误。然而，最后一行就遇到了麻烦。这也是一条有两处错误的消息，也是前两位数出错。但出错的两位数的值不同，而且这条有两处错误的消息的校验和恰好也是8——和原始校验和一样！因此接收这条消息的人将不能侦测到消息中有错误。

幸运的是，我们可以通过对校验和把戏进行一些微调来解决这个问题。第一步是定义一种新的校验和。让我们称这种新的校验和为“阶梯”校验和（staircase checksum），因为这个名字有助于通过想象在爬楼梯进行计算。想象你处于一个楼梯的底部，楼梯台阶编号为1、2、

3……依此类推。要计算阶梯校验和，你像之前一样把数字相加，但每个数都要和该数字所在位阶数相乘，每个数都比前一个数大一个位阶。最后，你只保留最后一位数，和简单校验和一样。因此，如果消息是:

4 6 7 5 6

和之前类似，阶梯校验和通过首先计算阶梯和来结算:

（1 × 4） + （2 × 6） + （3 × 7） + （4 × 5） + （5 × 6）

= 4 + 12 + 21 + 20 + 30

= 87

然后只保留最后一位数，也就是7。因此“46756”的阶梯校验和为7。

所有这一切想说明什么？如果你同时使用简单校验和及阶梯校验和，那么你就保证能侦测到任何消息中两处错误。因此，用我们新的校验和把戏来传输原始消息会多出两个数字：首先是简单校验和，其次是阶梯校验和。比如，消息“46756”会以:

4 6 7 5 6 8 7

传输。当你收到消息时，你仍然必须提前知道会运用哪些把戏。但假设你确实知道，那么检查错误就像使用简单校验和一样容易。在这个例子中，你先把最后两个数（简单校验和8及阶梯校验和7）放一边，然后计算消息其余部分的简单校验和（46756的简单校验和是8），阶梯校验和也要计算（结果为7）。如果两个计算后的校验和值和发送的两个校验和匹配（这个例子中的两个值的确匹配），你就可以保证这条消息要么是正确的，要么至少有三处错误。

下一张表显示了这两种校验和的实际运用情况。这张表和前面那张表几乎一模一样，除了在每行末尾加上了阶梯校验和，以及新加了一行作为额外的例子。当消息中有一处错误时，我们发现这条消息的简单校验和及阶梯校验和均与原始消息的不同（简单校验和是5而不是8，阶梯校验和是4而不是7）。当消息中有两处错误时，有可能两个校验和值都不相同，如表中第3行的简单校验和是4而不是8，阶梯校验和是2而不是7。但正如我们已经发现的，有时候当消息中有两处错误时，该消息的简单校验和不会改变。表中第4行就是这种情况的一个例子，其简单校验和仍然是8。但因为阶梯校验和和原始消息的不同（9而不是7），我们仍然知道这条消息有误。在表中最后一行，我们能看到还有另外一种情况：这条含有两处错误的消息的简单校验和不同（是9而不是8），但其阶梯校验和却一样（7）。不过，这里的意思是我们仍然能侦测到错误，因为两个校验和中至少有一个和原始校验和不同。尽管需要用一些稍微技术性的数学来证明，但有一点是确认无疑的：只要错误不超过两处，你就都能够侦测到错误。

| | | | | | | 简单和阶校验和 | |
|---|---|---|---|---|---|---|---|
| 原始消息 | 4 | 6 | 7 | 5 | 6 | 8 | 7 |
| 有一处错误的消息 | 1 | 6 | 7 | 5 | 6 | 5 | 4 |
| 有两处错误的消息 | 1 | 5 | 7 | 5 | 6 | 4 | 2 |
| 有两处（不同）错误的消息 | 2 | 8 | 7 | 5 | 6 | 8 | 9 |
| 另一条有两处（不同）错误的消息 | 6 | 5 | 7 | 5 | 6 | 9 | 7 |

既然我们对基本方法有了初步了解，我们需要意识到，刚刚描述的校验和把戏只能保证在相对较短的消息上奏效（少于10位数）。但与此非常类似的概念能被应用在较长的消息上。通过一定的简单操作序列，如把数字加起来，将数字和其位阶相乘，或按照固定模式将一些数字交换，我们还是有可能定义校验和的。尽管这听起来很复杂，

计算机却能毫不费力地迅速计算这些校验和。结果证明，这些方法用在侦测消息中的错误上极其有效且实用。

上面描述的校验和把戏只生成两个校验和数字（简单校验和及阶梯校验和），但真正的校验和通常会生成比这长得多的数字——有时长达150位。（我在本章中讲的都是十进制数0~9，而不是二进制数0和1，二进制数在计算机通信中要更经常被用到。）重点是，校验和的长度（要么和上面的例子一样是2位，要么和在实际中运用的校验和一样，有近150位）是固定的。不过，尽管所有校验和算法生成的校验和长度都是固定的，只要你愿意，你都能计算消息的校验和。因此，对于非常长的消息来说，即便一个相对较大的校验和（如150位数），最终和消息本身相比也极小。比如，假设你从互联网上下载了一个20兆的软件包，你使用了一个100位数的校验和来验证它的正确性。软件包的校验和也比不上软件包大小的十万之一。我敢肯定，你会认为这个杂项水平可以接受！而数学家会告诉你，使用这种长度的校验和侦测错误，其失败的概率极其微小，在现实中几乎不可能失败。

和前面一样，这里也有几处重要的技术细节。并不是任何一个百位数校验和系统对失败都有如此高的抵抗性。这需要一种被计算机科学家称为加密哈希函数（cryptographic hash function）的特定校验和，尤其是在恶意敌人而非糟糕信道的随机变动对信息作出改变时。这是个非常现实的问题，因为也许就有一名邪恶黑客试图更改这个20兆的软件包，并让其具有相同的100位校验和数，而这个不同的软件其实是要控制你的计算机！使用加密哈希函数能消除这种可能性。

## 定位把戏

既然知道了校验和，现在我们可以回到最初同时侦测和纠正通信错误的问题上了。我们已经知道如何做到这一点，要么使用低效的重

复把戏，要么使用高效的冗余把戏。但还是让我们先回到这个问题，因为我们从未真正地了解代码字是如何创造的，而代码字则是这些把戏的关键成分。我们的确有用英语单词描述数字的例子，但那种代码字不如计算机实际使用的代码字高效。我们也看过一段汉明代码的真实例子，但没有解释一开始代码字是如何生成的。

因此，现在我们要学习另一套可能的代码字，用来执行冗余把戏。因为这种冗余把戏很特别，它能让你迅速定位一处错误，我们称其为“定位把戏”（the pinioint trick）。

正如我们在校验和把戏中做的一样，我们将和完全由数字0~9组成的消息打交道，但你要记住，这只是为了方便起见。将字母消息转译成数字很简单，因此在这里描述的技术能被应用到任何消息上。

为简单起见，让我们假设消息恰好有16个数字，但这并非该技术在现实中的应用。如果你有一条长消息，将其打碎成16位数长的块，并单独处理每块数据。如果消息比16个数字短，用0补成16位数。

定位把戏的第一步是重新排列消息中的16个数，将其排列成一个从左往右、自上向下读的方框。如果实际消息是:

4 8 3 7 5 4 3 6 2 2 5 6 3 9 9 7

重新排列为：

| | | | |
|---|---|---|---|
| 4 | 8 | 3 | 7 |
| 5 | 4 | 3 | 6 |
| 2 | 2 | 5 | 6 |
| 3 | 9 | 9 | 7 |

下一步，我们计算每一行的校验和，并添加在每行的右侧：

| | | | | |
|---|---|---|---|---|
| 4 | 8 | 3 | 7 | 2 |
| 5 | 4 | 3 | 6 | 8 |
| 2 | 2 | 5 | 6 | 5 |
| 3 | 9 | 9 | 7 | 8 |

这些简单校验和的计算方式和前面一样。比如，要得到第二行的校验和，你需要计算5+4+3+6 = 18，并保留最后一位数8。

定位把戏的下一步是计算每一栏的简单校验和，并添加在每列的底部：

| | | | | |
|---|---|---|---|---|
| 4 | 8 | 3 | 7 | 2 |
| 5 | 4 | 3 | 6 | 8 |
| 2 | 2 | 5 | 6 | 5 |
| 3 | 9 | 9 | 7 | 8 |
| 4 | 3 | 0 | 6 | |

和前面一样，求简单校验和的方法很清楚。比如，要得到第二行的校验和，你需要计算3+3+5+9 = 20，并保留最后一位数0。

定位把戏的下一步是重新排列所有数，让其能以一次一个数的方式存储或传输。你通过从左往右、自上向下读的方式读数。因此，最后你会得到如下24位数的消息：

4 8 3 7 2 5 4 3 6 8 2 2 5 6 5 3 9 9 7 8 4 3 0 6

现在，想象你已经收到了一条使用定位把戏传输的消息。你需要采取哪些步骤来得到原始消息，纠正其中出现的任何通信错误？让我们举个例子来说明。原始的16位数消息和上面的一样，但为了让事情变得更有趣，假设出现了一处通信错误，其中一个数被替换了。现在还不必担心被替换的数是哪个，我们很快就能运用定位把戏来确定。

假设你收到的24位数消息是：

4 8 3 7 2 5 4 3 6 8 2 7 5 6 5 3 9 9 7 8 4 3 0 6

你要做的第一步就是把数字放入一个五行五列的方框中，最后一行和最后一列都对应随原始消息一起发送的校验和数字：

| | | | | |
|---|---|---|---|---|
| 4 | 8 | 3 | 7 | 2 |
| 5 | 4 | 3 | 6 | 8 |
| 2 | 7 | 5 | 6 | 5 |
| 3 | 9 | 9 | 7 | 8 |
| 4 | 3 | 0 | 6 | |

接下来，计算每一行每一列四个数的简单校验和，在接收到的校验和值旁边新建一行和一列，并在其中记录计算结果：

| | | | | | |
|---|---|---|---|---|---|
| 4 | 8 | 3 | 7 | 2 | 2 |
| 5 | 4 | 3 | 6 | 8 | 8 |
| 2 | 7 | 5 | 6 | 5 | 0 |
| 3 | 9 | 9 | 7 | 8 | 8 |
| 4 | 3 | 0 | 6 | | |
| 4 | 8 | 3 | 6 | | |

这里会出现两组校验和值：你发送的，以及你计算出来的，记住这一点至关重要。在绝大多数情况下，两套值都一样。事实上，如果它们都一样，你可以得出结论，收到的消息极有可能是正确的。但如果出现一个通信错误，计算得出的一些校验和值会和发送的不同。注意，这个例子中有两个不同之处：第三行的5和0不同，第二列的3和8不同。冲突的校验和在框中被标注了出来：

| | | | | | |
|---|---|---|---|---|---|
| 4 | 8 | 3 | 7 | 2 | 2 |
| 5 | 4 | 3 | 6 | 8 | 8 |
| 2 | 7 | 5 | 6 | 5 | 0 |
| 3 | 9 | 9 | 7 | 8 | 8 |
| 4 | 3 | 0 | 6 | | |
| 4 | 8 | 3 | 6 | | |

关键在于：这些不同之处的位置正好说明了通讯错误出现的位置！错误肯定是在第三行（因为其他行的校验和都正确），错误也肯定是出现在第二列（因为其他列的校验和都正确）。正如你能在下图中看到的，这将错误的可能性圈定为一种——框中标出的7：

| | | | | | |
|---|---|---|---|---|---|
| 4 | 8 | 3 | 7 | 2 | 2 |
| 5 | 4 | 3 | 6 | 8 | 8 |
| 2 | 7 | 5 | 6 | 5 | 0 |
| 3 | 9 | 9 | 7 | 8 | 8 |
| 4 | 3 | 0 | 6 | | |
| 4 | 8 | 3 | 6 | | |

但这还不是全部，我们定位了错误，但还没纠正它。幸运的是，很容易做到这一点：我们只要用一个能让两个校验和都正确的数字替换出错的7即可。我们能看出，第二栏的校验和本应为3，但结果却是8。换句话说，校验和需要减5。我们把错误的7减5，得到2：

| 4 | 8 | 3 | 7 | 2 | 2 |
|---|---|---|---|---|---|
| 5 | 4 | 3 | 6 | 8 | 8 |
| 2 | 2 | 5 | 6 | 5 | 5 |
| 3 | 9 | 9 | 7 | 8 | 8 |
| 4 | 3 | 0 | 6 | | |
| 4 | 3 | 0 | 6 | | |

你甚至可以通过检查第三行来验证这一改变。第三行的校验值现在变成了5，和收到的校验和一致。错误同时被定位和纠正了！最后一步很明显，就是将纠正后的原始16位数消息从五行五列的框中取出，方法是从左往右，自上向下读（当然要忽略最后一行和最后一列）。结果是:

4 8 3 7 5 4 3 6 2 2 5 6 3 9 9 7

这和我们一开始用到的消息相同。

在计算机科学中，定位把戏的名称是“二维奇偶校验码”（two-dimensional parity）。奇偶校验码这个词的意思和简单校验和一样, 计算机在处理二进制数字时经常用到它。而奇偶校验码被形容为二维，是因为消息放在有两个维度的表格（行和列）中。二维奇偶校验码在一些真正的计算机系统中也有运用，但并不如其他一些冗余把戏高效。我在这里解释它的原因是，它能很容易地具化并展现在不要求复

杂数学的情况下，于当今计算机系统中的流行代码背后发现并纠正错误的风味。

## 现实世界中的纠错及侦错

纠错码于20世纪90年代问世，距离电子计算机诞生后不久。回顾一下不难看出原因：早期计算机相当不可靠，组件经常产生错误。但纠错码的真正根源出现得还要早，根源在电报和电话等通信系统中。因此，这两大导致纠错码发生的事件都发生在贝尔电话公司的研究实验室中也就不奇怪了。我们故事中的两位英雄克劳德·香农（Claude Shannon）和理查德·汉明都是贝尔实验室的研究人员。前面已经介绍了汉明：正是因为他对公司计算机在周末崩溃的愤怒，直接导致了第一批纠错码的发明，也就是现在闻名的汉明代码。

不过，纠错码只是一个更大专业的一部分，这个专业被称为信息理论学（information theory），而绝大多数计算机科学家将信息理论学的诞生归因于克劳德·香农在1948年发表的一篇论文。香农的一本自传中将这篇名为《通信的数学理论》（The Mathematical Theory of Communication）的卓越论文形容为“信息时代的大宪章”。欧文·里德（Irving Reed）这么形容这篇论文：“本世纪只有几件事能在对科学和工程学的影响力上超越（这篇论文）。通过这篇里程碑式的论文……他极大地改变了通信理论和实践的所有方面。”欧文·里德是里德—香农代码的共同发明人，我们将在下面提到里德—香农代码。为什么赞誉如此之高？香农通过数学展示了，有可能从根本上通过一个嘈杂的、引发错误的链接实现错误率极低的通信。直到数十年后，科学家们才在现实中接近实现了香农的理论最大通信率。

顺便说一下，香农的兴趣非常广泛。作为1956年达特茅斯人工智能大会（将在第六章末谈到）的四位主要组织者之一，他积极参与了

另一领域的创立：人工智能。这还没完。他还骑单轮车，并制造了一辆令人难以置信的单轮车。这辆单轮车有一个椭圆轮子，意味着车手随着单轮车前进可以上下移动！

香农的工作将汉明代码放到了一个更大的理论环境中，并为许多深入发展打下了基础。自此以后，汉明代码被用于一些最早期的计算机中，并仍广泛用于一些特定种类的内存系统中。里德—所罗门（Reed–Solomon）代码是另一类重要的代码。这些代码能被用来纠正每个代码字中的众多错误。[与这里的（7, 4）汉明代码截然不同，汉明代码只能纠正7位数代码字中的一个错误。]里德—所罗门代码基于一个名为有限域代数（finite field algebra）的数学分支，但你可以非常粗略地想象，它结合了阶梯校验和及二维定点把戏的特色。CD、DVD和计算机硬盘中都用到了里德—所罗门代码。

校验和在现实中的运用也很广泛，一般是用于侦测而非纠正错误。也许最常见的例子就算以太网了，现今地球上几乎所有计算机都用这一联网协议。以太网中应用了一种被称为CRC-32的校验和来侦测错误。最为普遍的互联网协议TCP也在其发送的每块（或包）数据上使用校验和。校验和不正确的包直接被丢弃了，因为TCP被设计用来在以后必要时自动传输这些包。发布在互联网上的软件包通常使用校验和验证：其中一种流行的校验和是MD5，另一种是SHA-1。这两种校验和都属于加密哈希函数，为抵御恶意篡改软件提供保护，也能防止随机通信错误。MD5校验和约有40位数长，SHA-1生成的数约有50位。SHA-1的同类校验和中有些抗错性甚至要更好，如SHA-256（约75位数）和SHA-512（约150位数）。

纠错及侦测代码这门科学不断壮大。自20世纪90年代开始，一种名为低密度奇偶校验码（low-density parity-check codes）的方法受到极大关注。这些代码如今的用途非常广泛，从卫星电视到通过深海光线进行的通信。下次你在周末享受高清卫星电视时，不妨遐思一下这个

令人回味的反讽：正是由于理查德·汉明在周末与早期计算机的斗争中产生了困扰，才有了我们现在周末的娱乐。

# 第六章　图形识别——从经验中学习

分析引擎没有原创任何东西的权利。它只能按照我们的指令执行任何事情。

——艾达·勒芙蕾丝（Ada Lovelace），
摘自她1843年有关分析引擎的笔记

在前面各章我们探索的领域里，计算机的能力都要远超人的能力。比如，计算机通常可以在一两秒内加密或解密一个大文件，而人通过手动计算执行相同的任务则需要数年时间。举个更极端的例子，想象一下，假设让人根据第三章中描述的算法手动计算数十亿网页的PageRank权重。这项任务太大，在现实中我们不可能完成这项任务。但互联网搜索公司的计算机在持续不断地执行这些计算。

而在本章，我们将审视一个人类具有天然优势的领域：图形识别领域。图形识别是人工智能的一部分，包括面部识别、物体识别、语音识别和笔迹识别等任务。更具体的例子，如判定一张照片是不是你姊妹的照片，或判定手写信封上的城市及州名。因此，图形识别可以更通俗地定义为，让计算机基于输入数据“聪明地”行动的任务，这些数据包含大量的变量。

给“聪明地”加双引号的理由很充分：计算机是否能展现出真正的智能，这个问题具有高度争议性。本章的开篇引言代表了这一争论中最早的论述之一：艾达·勒芙蕾丝于1843年就一台早期机械计算机的设计发表评论，这台早期计算机被称为分析引擎（Analytical Engine）。

但在这一声明中，她着重说明了计算机缺乏原创性：它们必须严格遵循人类程序员的指令。一段时间以来，计算机科学家们就计算机能否从根本上展现出智能争论不休。而如果把哲学家、神经科学家和神学家也算上的话，这一争论甚至会变得更复杂。

幸运的是，我们不必在本章解决机器智能的悖论。从我们的目的出发，我们最好还是将“智能”这个词换成“有用”。因此，图形识别的基本任务就是处理一些变量极多的数据——如不同光照环境下的各种人脸照片，或不同人书写许多不同字的笔迹样本——并做一些有用的事。毫无疑问，人能够智能地处理这些数据：我们能以令人难以置信的精确度识别人脸，能在不提前查看其他人笔迹样本的情况下，阅读几乎所有人的笔迹。计算机在处理这些任务上远不如人。但已经有一些精妙的算法出现，能让计算机很好地执行一些特定的图形识别任务。在本章，我们将学习三种这样的算法：最近邻分类器（nearest-neighbor classifier）、决策树（decision tree）以及人工神经网络（artificial neural network）。但首先，我们要对尝试解决的问题进行更科学的描述。

## 问题是什么？

从一开始看，图形识别的任务种类多得似乎超乎寻常。计算机能使用一个单独的图形识别技术工具包来识别笔迹、面部、语音及更多东西吗？这一问题的可能答案之一是：盯着我们的脸看，人脑在处理多种多样的识别任务上速度超快、精确度超高。我们能编写一个计算机程序来做到相同的事吗？

在讨论那样一个程序可能用到的技术之前，我们需要统一令人眼花缭乱的多种任务，定义一个单一问题让我们来尝试解决。要实现这一点，标准做法是将图形识别看作分类问题。我们假设要处理的数据

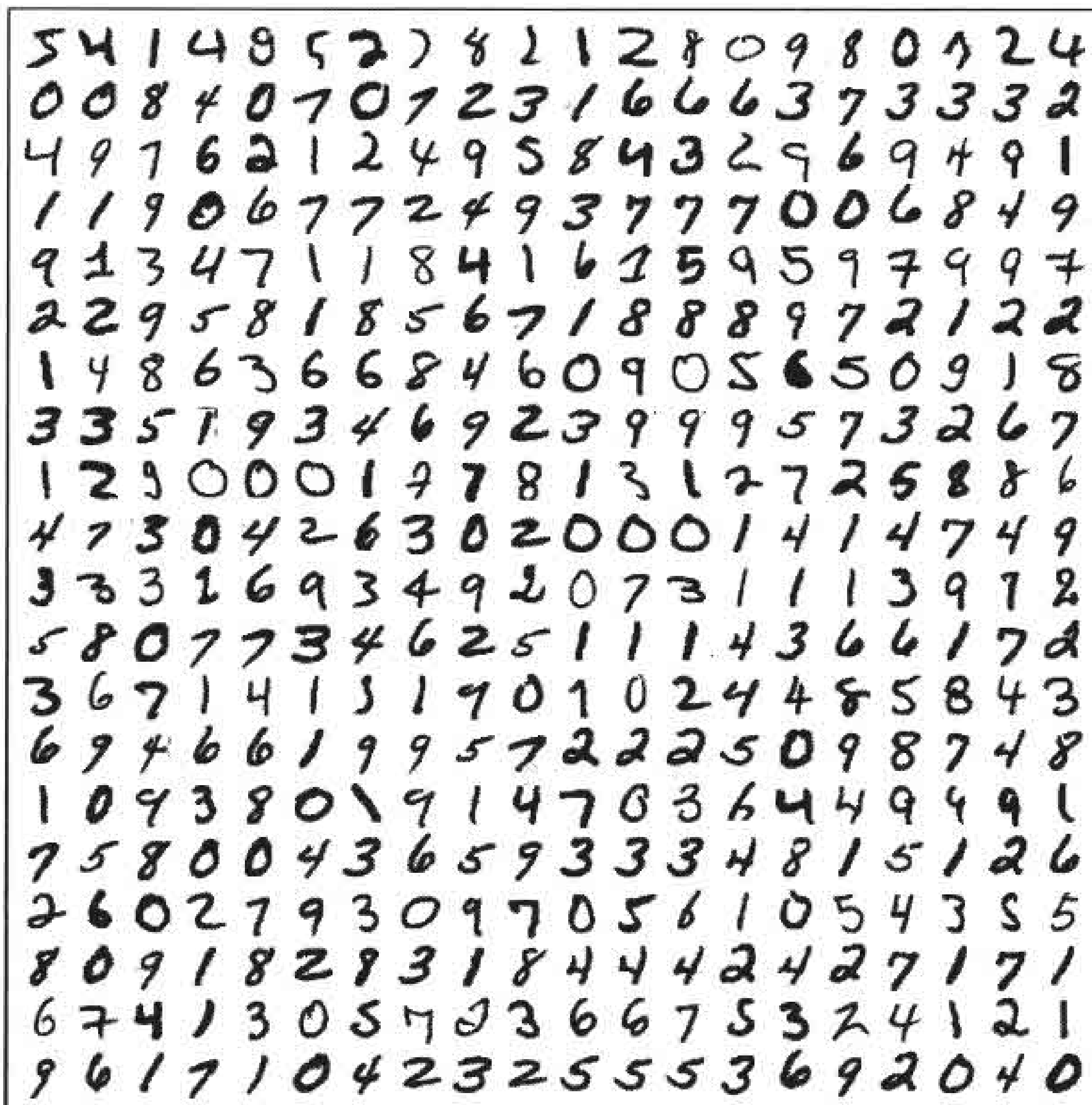
被分解成合理大小的块，这些块被称为“样本”，每一个样本都属于可能“类”（class）的固定集之一。比如，在一个人脸识别问题中，每一个样本都是一张人脸图片，而类则是系统能识别出的这些人的身份。有一些问题只有两个类。这方面的常见例子是对一种特殊疾病的医学诊断，而其中的两个类也许是“健康”和“得病”，这样每个数据样本就能组成单个病人的所有诊断结果（比如，血压、体重、X光照片以及其他很多方面）。因此，计算机的任务就是处理其从未见过的新数据样本，并将每个样本分到可能的一个类中。

具体来讲，接下来让我们集中精力于一个图形识别任务。这个任务是识别手写数字，其中一些典型样本在下文的图中有显示。这个问题里正好有10个类：数字0、1、2、3、4、5、6、7、8和9。因此，任务就是将手写数字样本分到其所属的类中。当然，这是个有着巨大实际影响的问题，因为美国和其他国家都使用数字邮编表示地址。如果计算机能快速且精确地识别这些邮编，机器分类信件的速度就要比人类高效得多。

很显然，计算机并没有内建手写数字是什么样的知识。事实上，人类也没有这种内建知识：通过结合其他人的详尽教授以及观看我们用来教授自己的例子，我们学会了如何去识别数字及其他手写符号。这两种策略（详尽教授和从例子中学习）也被用于计算机图形识别。不过，结果显示，即便是最简单的任务，详尽教授计算机也非常低效。比如，可以将我房子中的环境控制看作一个简单的分类系统。数据样本包含当前温度和时间，三个可能的类分别是“开启供暖”、“开启空调”和“两者均关闭”。因为我白天在办公室工作，我在白天就将系统设置为“两者均关闭”，而在这些时间之外，如果温度过低就会“开启供暖”，如果温度过高就会“开启空调”。因此，在设置温度调节器的过程中，我在某种程度上“教授”了系统针对这三个类执行分类。

不幸的是，没人能详尽地“教授”计算机解决更有趣的分类任务，如下一页的手写数字。因此计算机科学家们转向另一种策略：让计算机自动“学习”如何分类样本。基本策略是给计算机大量标记数据（labeled data）：已经被分类的样本。后面的图就给出了一些用于手写数字任务的标记数据。因为每个样本都带有一个标签（也就是它的类），计算机能运用多种分析把戏提取出每个类的特性。之后再向计算机提供一个未标记的样本，通过选择和未标记样本特性最接近的样本，计算机能推测未标记样本的类。

学习每个类特性的过程通常被称为“训练”，而标记数据则是“训练数据”。因此，概括来说，图形识别任务分为两个阶段：首先是训练阶段，计算机基于一些标记训练数据学习类；其次是分类阶段，计算机对新的未标记的数据样本进行分类。

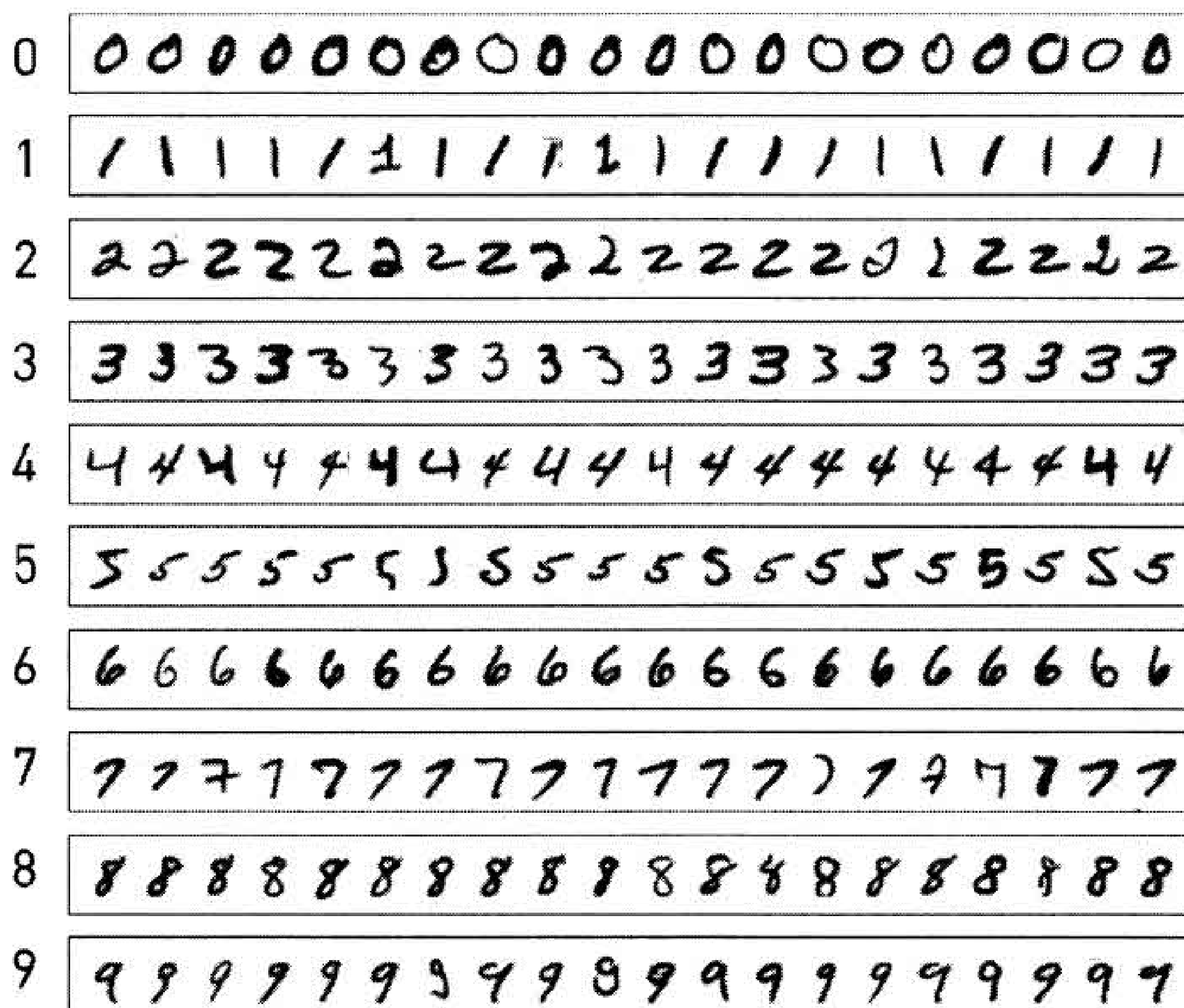


可以说绝大多数图形识别任务都是分类问题。在这个例子中，计算机的任务是将每个手写数字分到数字0、1、……、9的一个类中去。
数据来源：乐康（LeCun）等人的MNIST数据。

## 最近邻分类把戏

下面这个分类任务很有趣：你能只根据一个人的家庭住址，预测那个人会向哪个政治党派捐赠吗？很显然，这个例子中的分类任务不

能以完美的精确度执行，即便是人也不行：一个人的地址并不能透露足够多的信息让你预测这个人的政治倾向。不过，不管怎样，我们都愿意训练一个分类系统，让其仅仅根据房屋地址，预测这个人最有可能向哪个政党捐赠。

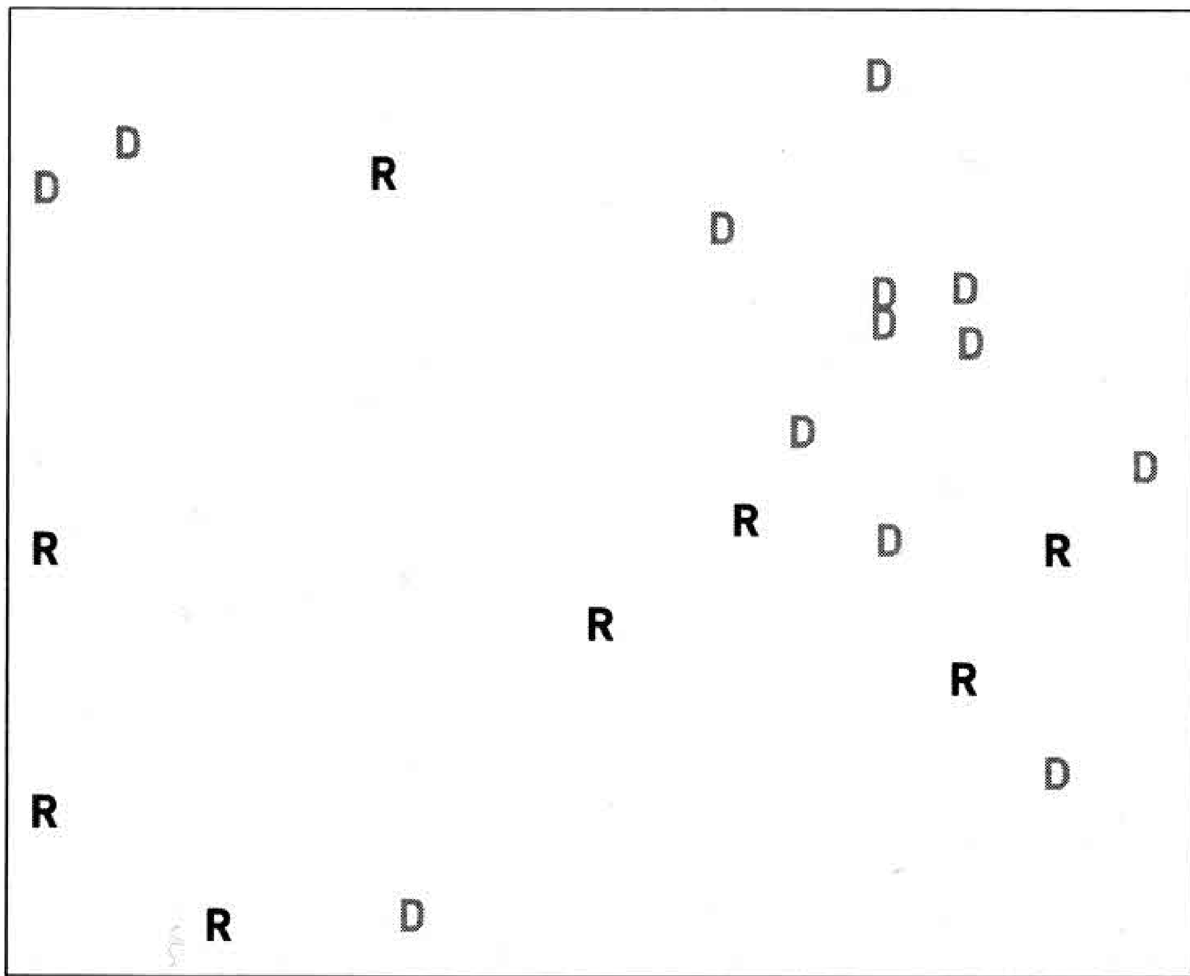


要训练一个分类器，计算机需要一些标记数据。在这个例子中，每个数据样本（一个手写数字）都带有一个表明其所属具体数字的标签。标签在左边，训练样本在右边框内。数据来源：乐康（LeCun）等人的MNIST数据。

下页的图显示了一些可能被用于这一任务的训练数据。这张图是2008年美国总统大选中堪萨斯州一个社区居民实际捐赠的地图。（也许你会对这个社区的名字感兴趣，这是堪萨斯州威奇塔市学院山社区。）为便于辨认，地图上并没有显示街道名，但捐赠了的每栋房屋

的实际地理位置都在地图上进行了精确的标记。向民主党人捐赠的房屋被标记为“D”，向共和党人捐赠的房屋被标记为“R”。

训练数据如此之多。当有一个新样本需要被分为民主党人或共和党人时，我们要怎么做？下面的图具体地展示了这个例子。训练数据和上一幅图中的一样，但另外添加了两处用问号表示的新地点。让我们首先关注上方的问号。只是瞥一眼，不用尝试做任何科学的事，你觉得这个问号最后可能属于哪一类？这个问号似乎被民主党人捐赠所围绕，因此“D”似乎是个相当有可能的选项。那另一个在左下方的问号呢？这个问号并没有完全被共和党人捐赠围绕，但其所在位置更像是在共和党人领地而非民主党人领地内，因此“R”会是个好推测。



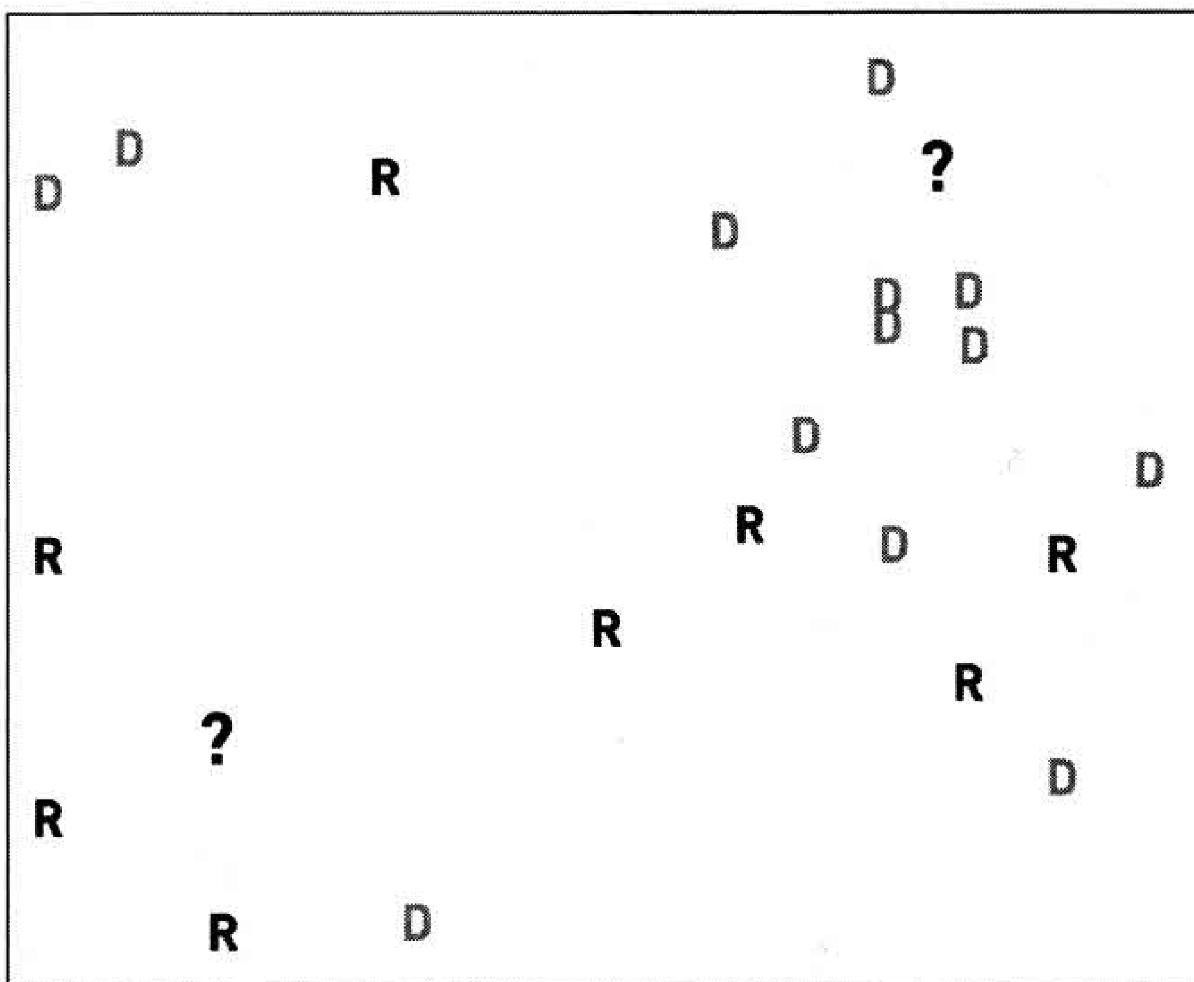

用于预测政党捐赠的训练数据。“D”代表向民主党人捐赠的房屋，“R”代表向共和党人捐赠的房屋。

数据来源：《哈芬顿邮报》，选金竞赛项目（Fundrace project）。

不管你是否相信，我们刚刚掌握了迄今为止发明的最强大、最有用的图形识别技术之一：一种被计算机科学家称为最近邻分类器的方法。“最近邻”把戏做的事，正如其名字所表达的含义一样。当你获得一个未分类的数据样本时，首先在训练数据中寻找该样本的最近邻，其次将最近邻所属的类作为你的预测。在下图中，这就是推测离每个问号最近的字母。

这一把戏另一个稍微成熟的版本被称为“K个最近邻”（K nearest-neighbors），K代表3或5这样的小数字。在这张图中，你要检查问号的K个最近邻，并选择在这些邻近对象中最受欢迎的类。我们可以在下图中看到实例。在这里，离问号最近的一个相邻对象是共和党人捐赠，因此最近邻把戏最简单的形式会将这个问号归为“R”。但如果我们转而使用3个最近邻方法，我们发现其中有两家是民主党捐赠，一家是共和党捐赠——在这一特定组合的社区中，民主党捐赠要更受欢迎，于是问号被归为“D”。

那么，我们该使用多少相邻对象呢？答案取决于要处理的问题。一般来说，实际工作者会尝试几个不同的值，看哪个值效果最好。这听起来也许不科学，但却反映了有效图形识别系统的现实。有效的图形识别系统一般是通过组合运用数学洞见、良好的判断及实际经验创造出来的。

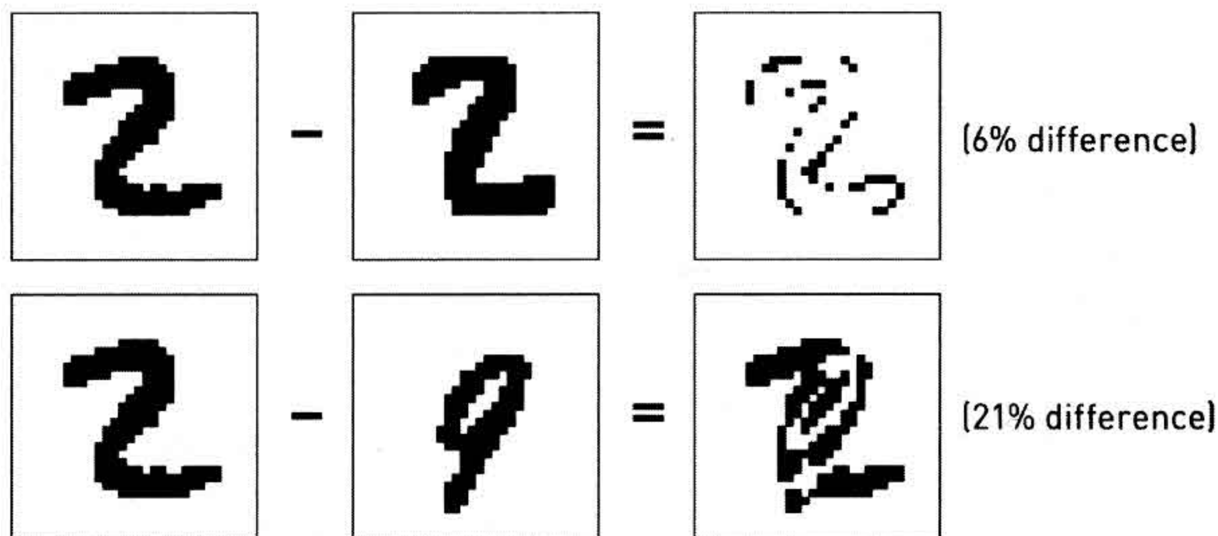



使用最近邻把戏进行分类。每个问号都被归为其最相邻对象所属的类。上面的问号是“D”，而下方的问号则是“R”。
数据来源：《哈芬顿邮报》，选金竞赛项目（Fundrace project）。

## 不同类型的“最近”邻

到目前为止，我们所着手的问题是经过特意挑选的，以便简单、直观地解释一个数据样本成为另一个数据样本“最近”邻的含义。因为每个数据点都在地图上标示了出来，我们只需使用点与点之间的地理距离，就能得出相距最近的点。但在每个数据样本都是像前文图中的手写数字时，我们该怎么办呢？我们需要一些方法来计算两个不同手写数字示例之间的“距离”。下面的图就展示了这样一种方法。

这一方法的基本理念是衡量数字图像之间的区别度，而非它们之间的地理距离。区别度以百分比形式衡量——区别度只有1%的图像是非常相近的邻，而区别度在99%的图像则相差很远。图中给出了具体例子。（在图形识别任务中，输入的信息会进行一些预处理步骤，这种情况很普遍。在本例中，每个数字都被调整为同一大小，并位于图像中心。）在图中最上方的一行中，我们能看到两张不同的手写2图像。通过对这些图像进行某种程度的“减法”，我们得到了右边的图像。这张图像的其余地方都是白色，只有这两张图像不同的少数地方为黑色。结果显示，这张区分度图像只有6%的地方是黑色的，因此这两个手写2图像是相对近邻。另一方面，在图中最下方的一行，我们能看到不同数字图像（一个2和一个9）削减后的结果。右边的区分度图像有众多黑像素点，因为这两张手写数字图像在很多地方不同。事实上，这张区分度图像有21%的地方是黑色的，因此这两张手写图像并非特别临近的邻。




计算两个手写数字的“距离”。在每一行，都要用第一张图减去第二张图，并得出右边的新图像，新图像中突出了这两张图中的区别。区分度图像中突出部分所占的百分比，就能被视为原始图像之间的“距离”。
数据来源：乐康（LeCun）等人的MNIST数据，1998年。

现在我们了解了如何计算手写数字之间的“距离”，那么为它们建立一个图形识别系统就容易了。我们从大量训练数据——就像前文图

中的手写数字一样，但例子要多得多——开始。这类图形识别系统的标准系统可能会使用十万个标记例子。当给予系统一个未标记的新手写数字时，系统会在十万个例子中搜寻，以找到一个和被分类的例子最接近的邻。记住，当我们在这里说“最近邻”时，我们真正说的是最小的区分度，通过上图中的方法计算。未标记数字会被打上和最近邻一样的标签。

结果表明，使用这种“最近邻”距离方法的系统效果相当好，精确度接近97%。研究人员们投入了巨大精力去寻找更加成熟的“最近邻”距离定义。运用最先进的距离衡量法，最近邻分类器在手写数字上的精确度能超过99.5%。这一精确度能和复杂得多的图形识别系统相比，比如名字很好听的“支持向量机”（support vector machine）和“回旋神经网络”（convolutional neural network）。最近邻把戏是计算机科学的一个真正的奇迹，它将赏心悦目的简洁性和令人印象深刻的高效性结合在了一起。

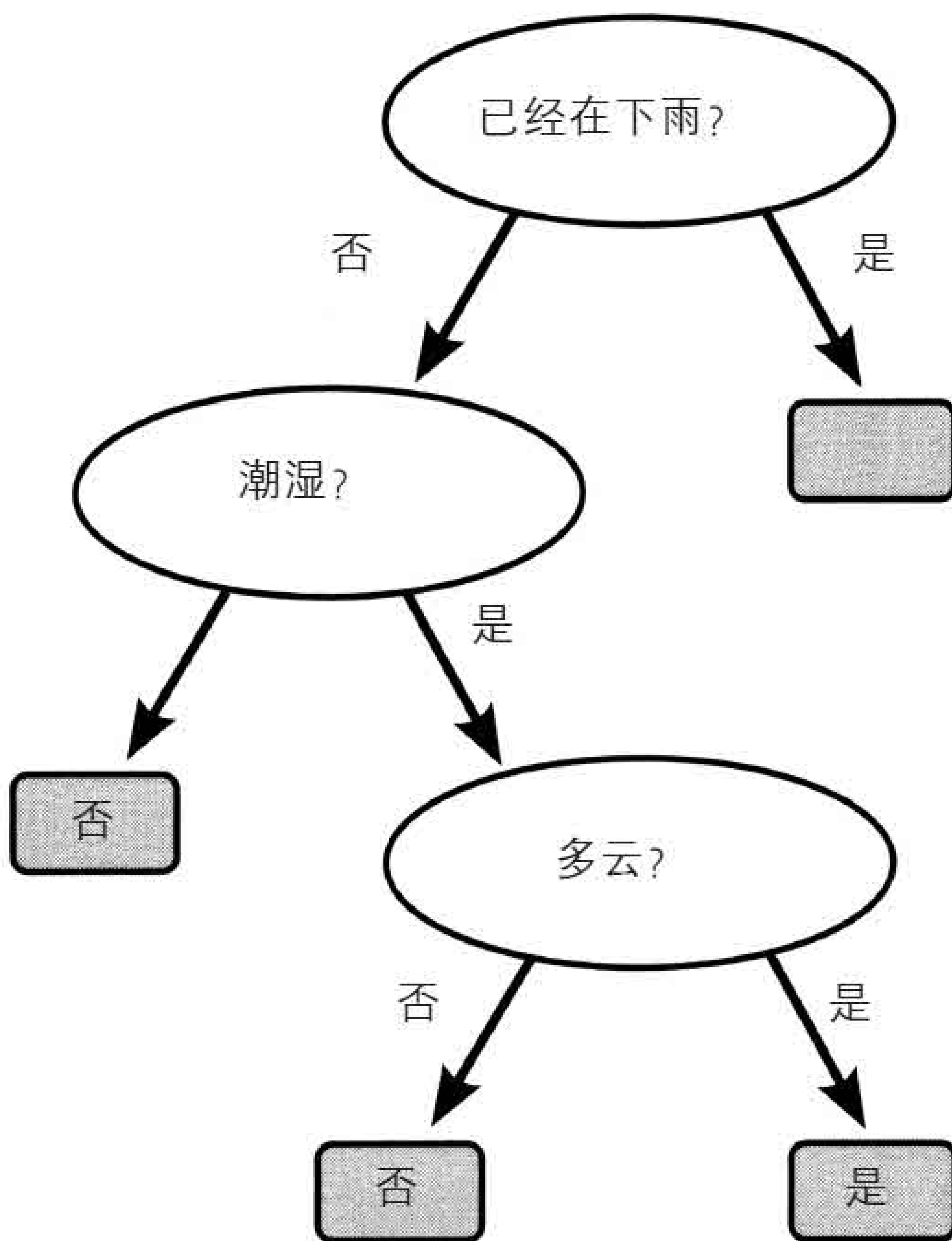
之前我曾强调，图像识别系统分为两个阶段运行：学习（训练）阶段会处理训练数据，提取类别中的一些特性；分类阶段会为未标记的新数据分类。那么，我们此前审视的最近邻分类器的学习阶段呢？似乎尽管我们获得了训练数据，却并未费力气去从中学得任何东西，而是直接使用最近邻把戏跳到了分类阶段。这正是最近邻分类器的特殊属性：它们无须任何详尽的学习阶段。在下一部分，我们将研究另一种分类器，在其中学习扮演一个远为重要的角色。

## 20个问题把戏：决策树

“20个问题”游戏对计算机科学家有一种特别的吸引力。在这个游戏中，一名玩家想着一个物体，而其他玩家必须依靠不超过20个是非问题的答案猜测这个物体的身份。你甚至能买到可以和你玩20个问题

游戏的小型手持设备。尽管这个游戏大多数时候被用于取乐儿童，但成年人玩这个游戏也能获得令人惊讶的好处。在玩这个游戏一段时间后，你开始意识到有“好问题”和“坏问题”。好问题肯定能给你大量“信息”（不管这些信息意味着什么），而坏问题则不能。比如，第一个问题问“它是铜做的吗”，就是个坏主意，因为答案是“不是”，可能性的范畴只被缩小了一点点。这些有关好问题和坏问题的直觉处于一个极有吸引力的领域——信息理论学——的核心。它们同样也是一种简单、强大的图形识别技术——决策树——的核心。

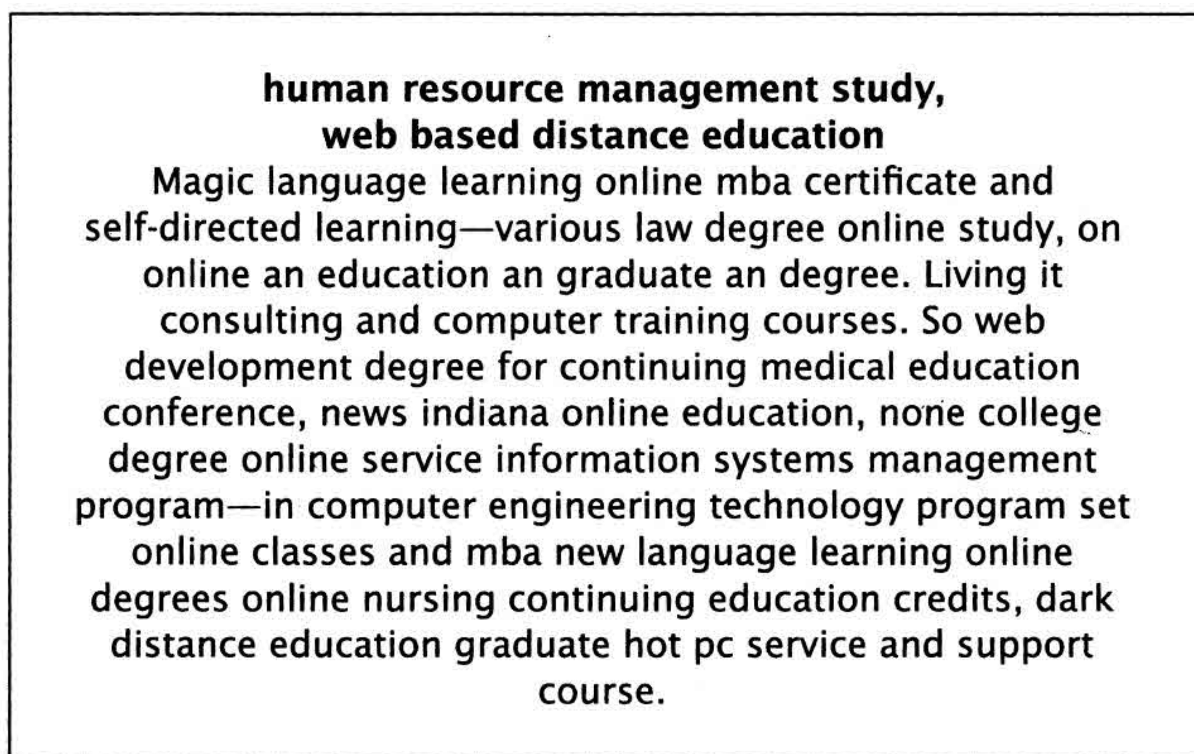
决策树基本上就是一个提前计划的20个问题游戏。下面的图显示了一个小例子。这是一个决定是否带伞的决策树。你只要从决策树顶部开始，按照问题的答案一路往下即可。当你到达决策树底部的一个框时，你也就得到了最终结果。




“我该带伞吗”的决策树。

你也许在想，这和图形识别及分类有什么关系？答案是，如果你有足够多的训练数据，系统可能会学习一个能进行精确分类的决策树。

让我们来研究一个例子，这个例子基于一个人们知之甚少但却极其重要的问题——网络垃圾。我们在第三章中提到过它，一些肆意妄为的网站操作人员试图通过人为制造链向特定页面的超链接，操纵搜索引擎的排名算法。这些狡诈的网站管理员使用的一个相关策略，就是创建拥有特别修饰内容，但对人一点用处都没有的网页。你可以在下页的图中看到一小段取自一个真实网络垃圾页面的内容。注意，那张图中的文本没有任何意义，但却重复列出了和在线学习有关的流行搜索术语。这个网络垃圾网页是为了提高其链向的特定在线学习网站的排名。

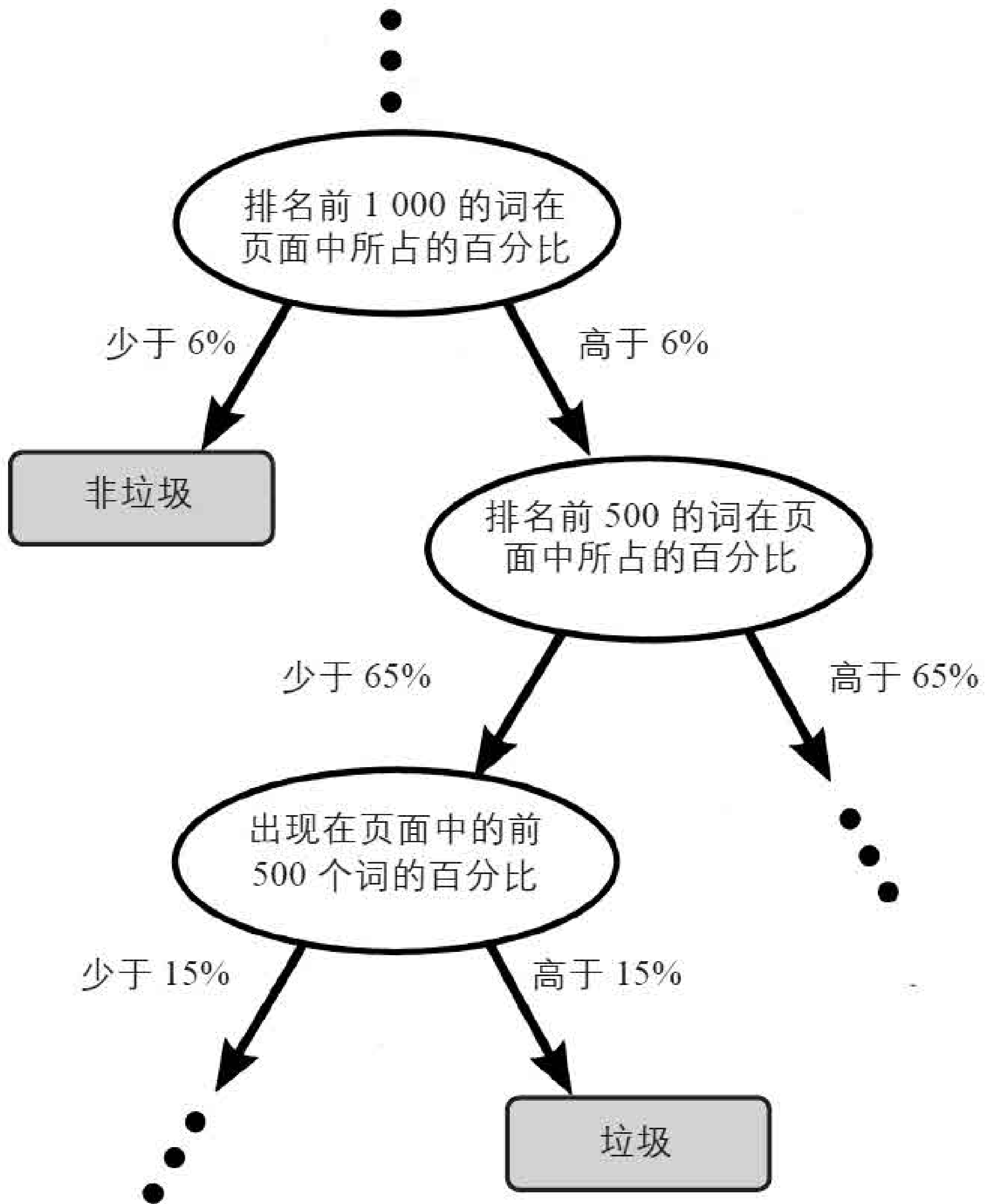



节选自一个“网络垃圾”页面。这个页面没有任何对人类有用的信息——其唯一目的就是操纵网络搜索排名。
资料来源：乐陶拉斯（Ntoulas）等人，2006年。

自然，搜索引擎要花费大量力气去尝试辨别和消除网络垃圾。这也是图形识别的完美应用：我们能取得大量训练数据（在网络垃圾的例子中就是网页），手动将它们标记为“垃圾”或“非垃圾”，并训练某种分类器。这正是微软研究院的一些科学家在2006年做的事。他们发

现，对这个特殊问题效果最好的分类器是人们喜爱已久的决策树。下面的图展示了他们得出的决策树的一小部分。

尽管完整的决策树依赖于许多不同的属性，下面的图展示的部分决策树专注于页面内文字的流行度。网络垃圾制造者喜欢在页面中加入大量流行词，以提升他们网页的排名，因此流行词占比较小，也预示该网页是垃圾的概率较低。这解释了决策树的第一个决策，其余决策遵循类似的逻辑。这个决策树的精确度在90%左右——离完美相差很远。但不管怎样，这是一个对抗网络垃圾制造者的无价武器。




辨别网络垃圾的决策树的一部分。点代表部分决策树出于简洁的目的省略了。
资料来源：乐陶拉斯（Ntoulas）等人，2006年。

要重点理解的不是决策树本身的细节，而是这一事实：整个决策树是由计算机程序基于约1.7万个网页上的训练数据自动生成的。这些“训练”页面被真人分为“垃圾”或“非垃圾”。好的图形识别系统需要投入巨大的人力，但这是一次性投入，并能产生长期回报。

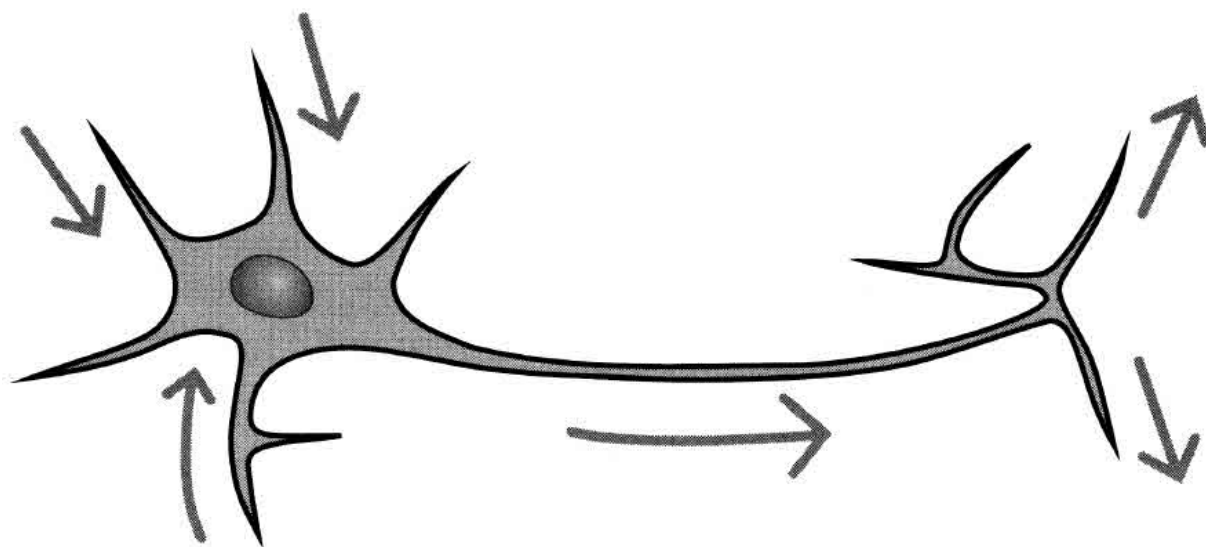
和我们之前讨论过的最近邻分类器相反，决策树分类器的学习阶段非常庞大。这个学习阶段如何进行？其主要思想和规划好一个20个问题游戏一样。计算机测试大量可能的第一个问题，寻找一个能得出最佳的可能信息的问题。然后，计算机将训练样本分为两组，取决于样本对第一个问题的答案，并产生第二个对这两个组都最佳的可能问题。计算机一直用这种方法往决策树底部前进，永远基于达到决策树某一点的训练示例集合来决定最佳问题。如果示例集合在某一点变得“纯净”，也就是说，集合中只包含垃圾页面或非垃圾页面——计算机就能停止生成新问题，输出对应剩余页面的答案。

总之，决策树分类器的学习过程可以很复杂，但它是全自动的，而且你只需要做一次。在此之后，你就得到了自己需要的决策树，而分类阶段则极其简单：就像20个问题游戏一样，你按照问题的答案向决策树底部移动，直到抵达结果框。基本上，只需要少数问题，因此分类阶段极度高效。同最近邻方法相比，最近邻方法在学习阶段无须费力，但分类阶段要求我们将需要分类的每个东西和所有训练示例进行比对（手写数字任务中的示例数是10万个）。

在下一部分，我们会碰到神经网络：它也是一种图形识别技术，其学习阶段不仅重要，而且直接受到人类和其他动物从环境中学习的方法的启发。

## 神经网络

从第一台数字计算机的创造开始，人脑令人印象深刻的能力就吸引并启发了计算机科学家。最早使用计算机模拟人脑的讨论之一，是由英国科学家阿兰·图灵（Alan Turing） 发起的，图灵还是一名顶级的数学家、工程师和译电码专家。图灵于1950年发表的经典论文《计算机器与智能》（Computing Machinery and Intelligence），以其对计算机是否能伪装成人类的哲学探讨而闻名于世。



一个标准生物神经元。电子信号按图中箭头所示方向流动。只有在输入信号足够大的情况下，神经元才会传输输出信号。

这篇论文介绍了一种评估计算机和人之间相似性的科学方法，也就是如今众所周知的“图灵测试”（Turing Test）。但在这篇论文的一段不怎么知名的文字里，图灵直接分析了使用计算机模拟人脑的可能性。他估计只需要十几G（千兆字节）的内存就足够了。

60年后，人们普遍认为图灵极大地低估了模拟人脑所需的工作量。自那篇论文发表以后，计算机科学家们使用了多种不同的方法寻求达成这一目标。其中一个结果就是人工神经网络领域（artificial neural networks），简称为神经网络。

## 生物神经网络

为帮助我们理解人工神经网络，我们首先要对真实的生物神经网络如何运行有大致了解。动物大脑由神经元细胞构成，每个神经元和其他许多神经元相连。神经元能通过这些连接发送电子和化学信号。其中一些连接被建立用于接收来自其他神经元的信号；余下的连接则向其他神经元传输信号（见上图）。

描述这些信号的一种简单方法是，在任何时刻，一个神经元要么“闲置”（idle），要么“开火”（firing）。当它闲置时，神经元并没有传输任何信号；当它开火时，神经元通过其所有外向连接频繁发送信号脉冲。神经元如何决定何时开火呢？这取决于其接收到的信号的强度。基本上，如果所有传入信号的总强度足够大，神经元就会开始开火；否则，神经元仍会闲置。粗略地来说，神经元会将其接收的所有输入“相加”，在总和足够大时开火。这一描述的一个重要修饰之处是，神经元实际上有两种输入，分别为兴奋性输入（excitatory）和抑制性输入（inhibitory）。兴奋性输入的强度会相加，而抑制性输入则会从总数中减去，因此一个强烈的抑制性输入倾向于阻止神经元开火。

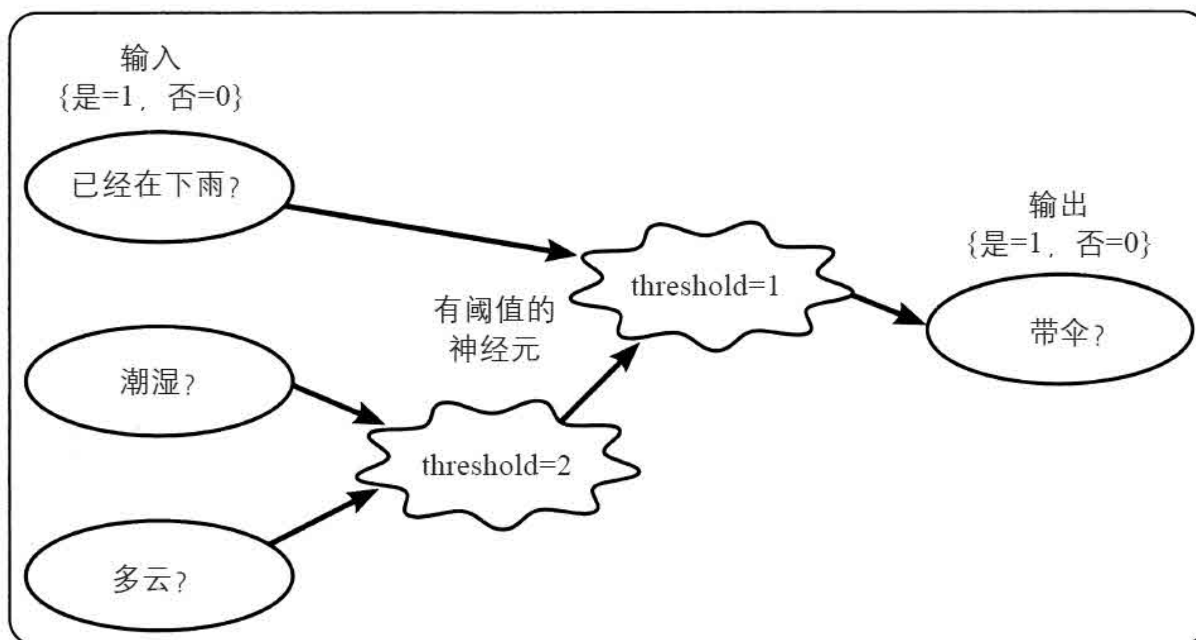
## 一个解决带伞问题的神经网络

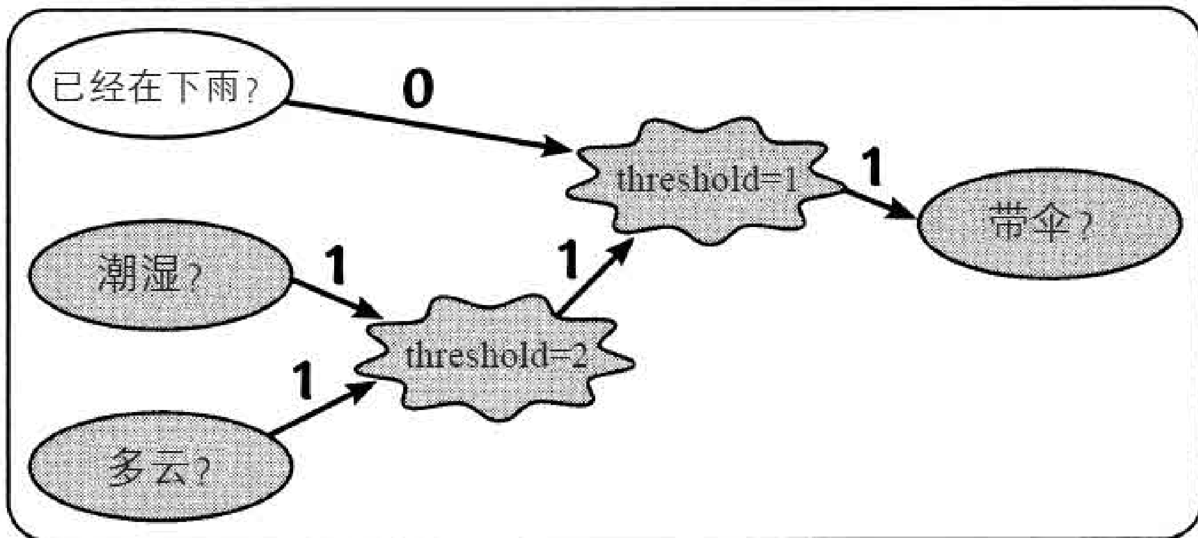
人工神经网络就是一个代表一小部分人脑的计算机模型，运算高度简化。我们在一开始会讨论一个人工神经网络的基础版本，这个基础版本在处理之前思考的带伞问题上效果良好。之后，我们将运用一个拥有成熟功能的神经网络来处理“太阳镜问题”。

我们基础模型中的每个神经元都有一个号码，这个号码被称为神经元的阈值（threshold）。当模型运行时，每个神经元都将其接收的输入相加。如果输入总量大于或等于阈值，神经元就开火，否则就仍

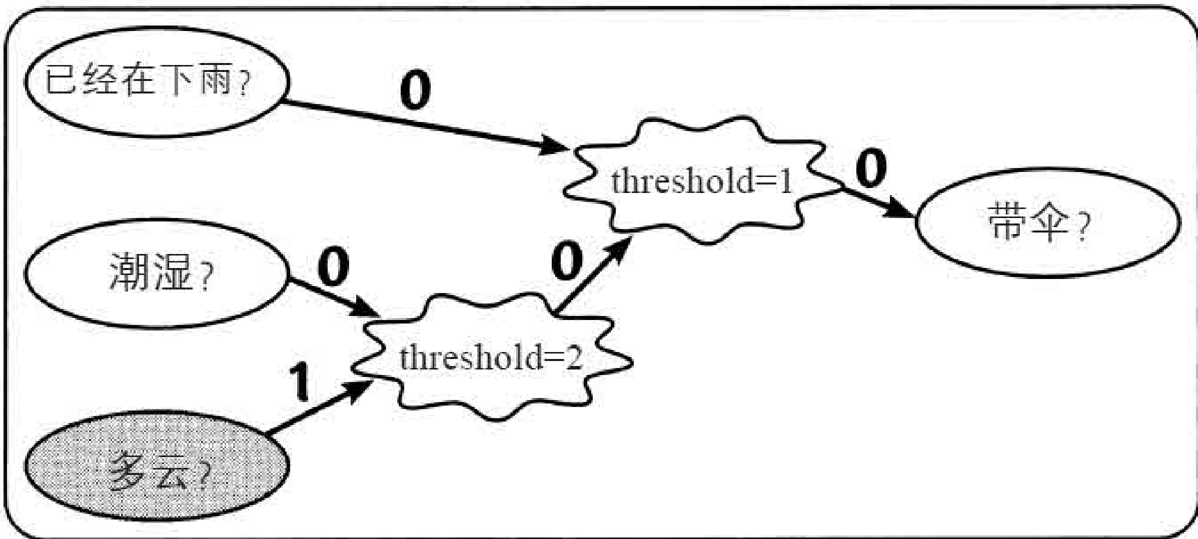
然闲置。下图就展示了一个神经网络，处理我们之前思考的极其简单的带伞问题。在图的左边，有三个输入进入网络。你可以将这些输入想象为动物脑中的感觉输入。和我们的眼睛及耳朵触发并发送至脑部神经元的电子及化学信号一样，图中三个输入向人工神经网络中的神经元发送信号。这个网络中的三个输入都是兴奋性输入。如果相对应的情况是真的，那么每个输入都会传输强度+1的信号。比如，如果现在多云，那么被标记为“多云”的输入，会发送一个强度+1的兴奋性信号；反之，它不会发送任何东西，也就相当于一个强度为0的信号。

如果我们忽略输入和输出，这个网络只有两个神经元，每个神经元的阈值都不同。湿润度和云量输入的神经元，只有在两个输入都活跃时才开火（假设其阈值为2），而另一个神经元只要输入之一是活跃的就能开火（假设其值为1）。这样做的效果在下图中展示了出来，你能在其中看到最终输出是如何根据输入改变的。






潮湿多云，但未下雨。




多云，但既不潮湿也没下雨。

最上方的图：一个解决带伞问题的神经网络。下面两张图：运行中的带伞神经网络。“开火”的神经元、输入和输出都加粗了。在中间的图中，输入状态是未下雨，但却既潮湿又多云，产生了带伞的决策。在最下面的图中，唯一的活跃输入是：“多云？”结果是不带伞的决策。



将由一个神经网络“识别”的人脸。事实上，与识别人脸不同的是，我们将处理比识别人脸更简单的问题，判断一张人脸上是否戴了太阳镜。来源：汤姆·米切尔（Tom Mitchell），《机器学习》（Machine Learning），麦格劳–希尔出版社（1998年）。感谢授权使用该图像。

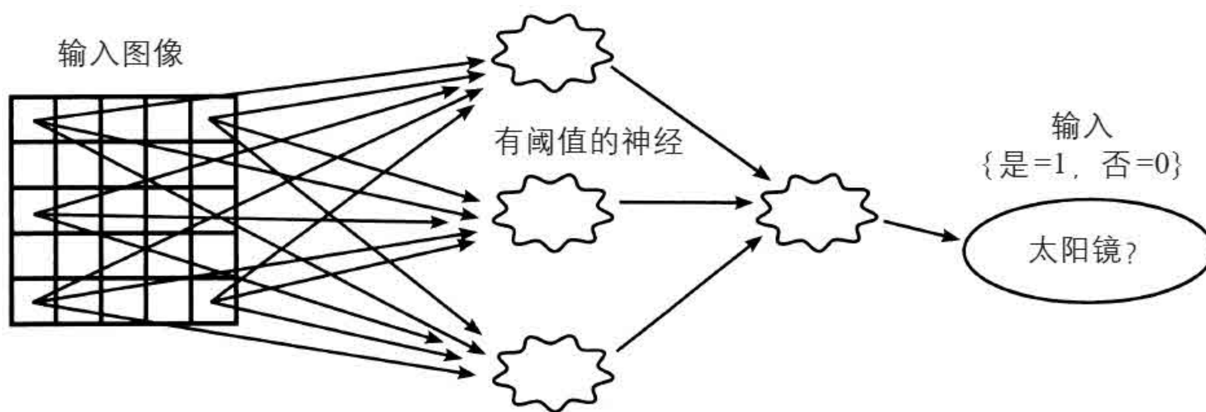
到这里，你也许该回头再看看前文中用于解决带伞问题的决策树。结果显示，当给予相同的输入时，决策树和神经网络产生了相同的结果。对于这个非常简单的人为问题而言，决策树作为代表很可能要更合适。但我们接下来将探究一个很复杂的问题，它将展示神经网络的真正力量。

## 一个解决太阳镜问题的神经网络

作为一个能通过使用神经网络成功解决的现实问题的例子，我们将处理被称为“太阳镜问题”的任务。这一问题的输入是一个低分辨率人脸照片的数据库。数据库中的人脸角度不一：有些人直视镜头，有些人的头仰着，有些人看向左边或右边，还有一些人戴着太阳镜。上图显示了其中一些例子。

在这里，我们故意使用低分辨率照片，以便让我们的神经网络容易描述。事实上，这些图片的宽和高都只有30像素。不过，我们很快就会知道，神经网络在如此粗糙的输入下也能得出出人意料的好结果。

这个人脸数据库可以通过神经网络来执行标准面部识别——即判定照片中人的身份，不管这个人是否看向镜头或戴太阳镜。但在这里，我们要处理一个更容易的问题，以更清晰地展示神经网络的属性。我们的目标是判定一张人脸是否戴太阳镜。




一个解决太阳镜问题的神经网络。

上图显示了这个网络的基本结构。这是一张概括图，并没有显示实际网络中使用的每个神经元或每个连接。这张图最明显的特点就是右边的那个输出神经元，如果图像含有太阳镜，这个输出神经元就产生一个1，反之则为0。在网络中央，有三个直接从输入图像接收信号并向输出神经元发送信号的神经元。这个网络最复杂的部分在图左边，我们可以看到从输入图像到中央神经元的连接。尽管并没有显示所有连接，实际网络中输入图像的每个像素和每个神经元都有连接。快速计算后，你会发现连接数相当庞大。记得我们使用的图像宽高都为30像素吧。因此，即便以现代标准来看这些图像很小，但每张图片也都包含30 × 30 = 900个像素。而图中有三个中央神经元，也就是说，这个网络的左边总共有3 × 900 = 2 700个连接。

这个网络的结构是如何决定的呢？这些神经元能按照不同的方式相连吗？是的，有许多种不同的网络结构能得出太阳镜问题的好结果。一个网络结构的选择，通常是基于之前运行良好的经验。和以前一样，我们再一次看到，和图形识别系统打交道需要洞见和直觉。

不幸的是，我们很快就会看到，在网络正确运行之前，我们选择的网络中的所有2 700个连接，都要按某种方式“调整”。我们如何才有可能处理这种调整数以千计不同连接的复杂度？答案是调整会通过学习训练示例自动完成。

## 增加加权信号

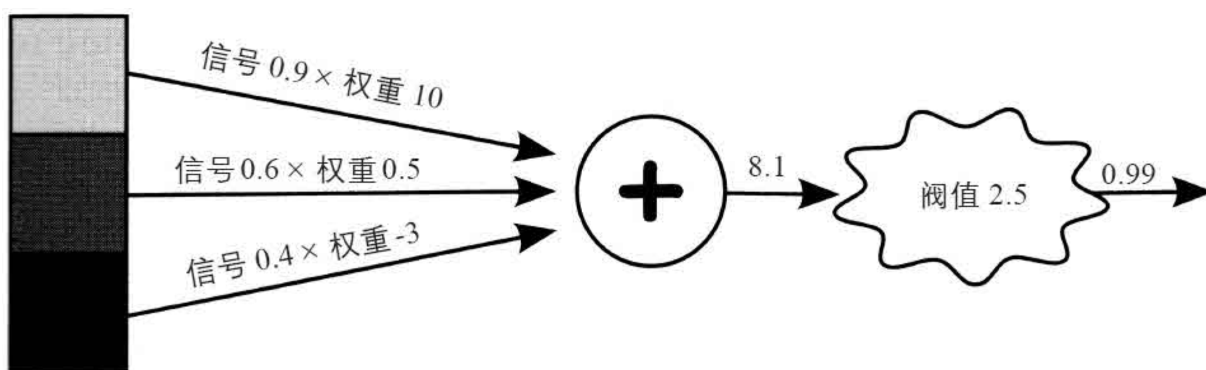
正如我之前提到的，我们用于解决带伞问题的网络是一个基础版人工神经网络。对于太阳镜问题，我们将添加三个重要的增强措施。

**增强措施 1：信号只能携带0和1之间的任意值。**这和带伞网络相反，带伞网络中的输入和输出信号被限制在0和1，不能携带任何中间值。换言之，新网络中的信号值可以是0.0023或0.755。为落实到具体，想想太阳镜例子。输入图像中一个像素的亮度对应通过那个像素连接输出的信号值。因此，一个全白的像素会发送值1，而一个全黑的像素会发送值0。不同灰度会对应地产生0~1的值。

**增强措施 2：输入总和通过加权求和计算。**在带伞网络中，神经元在没有用任何方法替换输入的情况下将输入相加。不过，在实际中，神经网络考虑了每个连接的强度不同。连接的强度由一个数字代表，这个数字被称为该连接的权重。大的正权重（如51.2）代表了强烈的兴奋性连接——当信号穿过这样的连接时，其下游神经元极有可能会开火。大的负权重（如–121.8）代表了强烈的抑制性连接——信号穿过这样的连接时，其下游神经元极有可能依然闲置。权重较小（如0.03或−0.0074）的连接对其下游神经元是否开火影响很小。（在现实中，权重被定义为“大”或“小”只是相对其他权重而言，因此，只有我们假设这些数字示例在同一神经元的连接上，这里给出的数字示例才有意义。）当一个神经元计算其输入的总和时，每个输入信号在

加进总和之前，都会和其连接的权重相乘。因此，大权重的影响力要比小权重的大，也让兴奋性和抑制性信号有可能相互抵消。

**增强措施 3：阈值的作用被软化。**阈值不再将神经元的输出限定在全开（比如1）或全关（比如0）上；输出可以是0和1之间（包含0和1）的任意值。当输入总和远低于阈值时，输出会接近于0，当输入总和远高于阈值时，输出接近于1。但当输入总和接近阈值时，会产生一个接近0.5的中间输出。比如，假设一个神经元的阈值是6.2，强度为122的输入可能会得到0.995的输出，这是因为输入远大于阈值。但强度为6.1的输入接近阈值，则其可能会产生一个0.45的输出。这种效果在所有神经元上都有，包括最终输出神经元。在我们的太阳镜应用中，这意味着，输出值接近1强烈暗示了存在太阳镜；而输出值接近0，则强烈暗示了不存在太阳镜。




信号在相加前会乘以一个连接权重。

上图展示了带有全部三种增强措施的新型人工神经元。这个神经元接收来自三个像素的输入：一个明亮像素（信号强度0.9）、一个中等明亮的像素（信号强度0.6）和一个颜色较深的像素（信号强度0.4）。这三个像素跟神经元连接的权重分别是10、0.5和−3。这些信号先和权重相乘再相加，得到这个神经元的输入信号总强度为8.1。因为8.1要远大于神经元的阈值2.5，因此输出非常接近于1。

## 通过学习调整一个神经网络

现在我们已经准备好去说明，调整一个人工神经网络究竟意味着什么。首先，每个连接（记住，可能会有成千上万个连接）都必须有权重值，这个值可以为正（兴奋性），也可以为负（抑制性）。其次，每个神经元都必须有个合适的值作为阈值。你可以将权重和阈值想象成网络中的小刻度盘，每个刻度盘都能像电灯开关中的调光器一样调上调下。

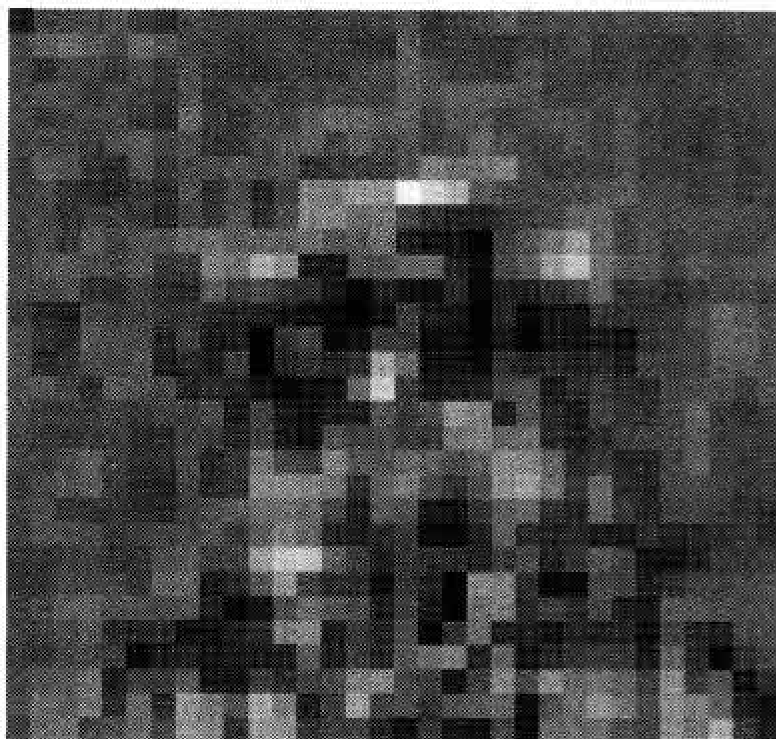
手动调节这些刻度盘需要耗费的时间会令人望而却步。相反，我们可以使用计算机在学习阶段设置这些刻度盘。最开始，这些刻度盘的值都是随机值。（这看起来也许非常随意，但专业人士在实际应用中也是这么做的。）之后，计算机会获得第一个训练样本。在这个例子中，第一个训练样本是一个可能戴也可能没戴太阳镜的人的照片。这个样本会通过这个网络处理，产生一个在0和1之间的输出值。然而，因为这个样本是训练样本，我们知道网络理应得出的“目标”值。关键点在于稍微调整一下网络，让其输出更接近于想要的目标值。比如，假设第一个训练样本恰巧有太阳镜。那么目标值就是1。因此，整个网络中的所有刻度盘都要稍微调节一下，以便让网络输出值朝目标值1的方面转移。如果第一个训练样本不包含太阳镜，每个刻度盘都要朝相反的方向移去一点点，以便让输出值向目标值0移动。你应该已经知道这个过程是如何进行的了吧。网络一个接一个地获得训练样本，每个刻度盘都要被调整以提升网络效能。在多次运行完所有训练样本后，网络的效能基本上会达到很高水平，而学习阶段也以将当时的刻度盘配置而结束。

实际上，如何计算刻度盘的这些细微调整的细节相当重要，但这需要一些超出本书范围的数学。我们需要的工具是多变量微积分（multivariable calculus），这一般是在大学中等数学课程中教授。是的，数学的确重要！还有，注意我在这里描述的方法，专家称之为“随

机梯度下降”（stochastic gradient descent），而这只是用于训练神经网络的多种公认方法之一。

所有这些方法都有相同的特点，让我们专注于大局吧：神经网络的学习阶段相当耗费精力，涉及对所有权重和阈值的反复调整，直到网络在训练样本上运作良好。不过，所有这些都能被计算机自动完成，而结果就是一个能简单高效地对新样本分类的网络。

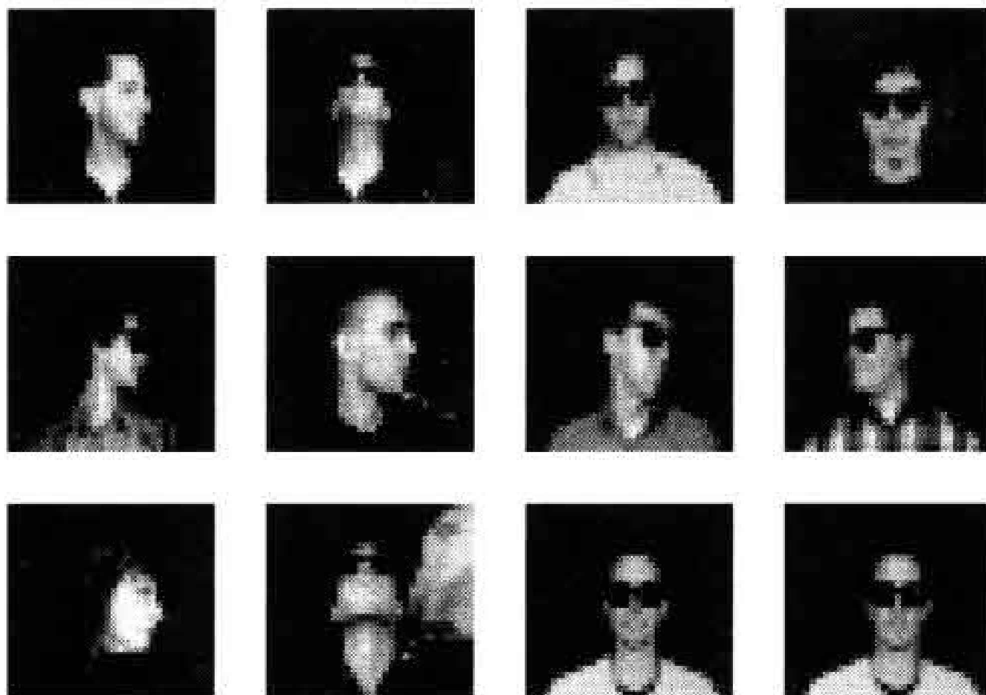
让我们来看看神经网络如何解决太阳镜问题。一旦学习阶段完成，从输入图像到中央神经元的成千上万个连接，都会被赋予一个数字权重。如果我们能将所有像素的连接集中到一个神经元上（比如最上方的），我们就能通过将它们转化为一张图片，从而非常方便直观地看到这些权重。权重的视觉化在下图中显示，图中所有像素都只连接到一个中央神经元上。在这张视觉化图中，强烈的兴奋性连接（比如大的正权重连接）显示为白色，而强烈的抑制性连接（比如大的负权重连接）显示为黑色。不同灰度的颜色代表中间强度的连接。每个权重都显示在与其相对应的像素点上。仔细看一下图。在太阳镜通常会出现的区域有一个非常明显的由强烈抑制性权重组成的地带——事实上，你几乎可以肯定这张权重图真的包含某副太阳镜的图像。我们可以称之为太阳镜的“幻影”，因为它们并不代表任何现实中存在的特定太阳镜。



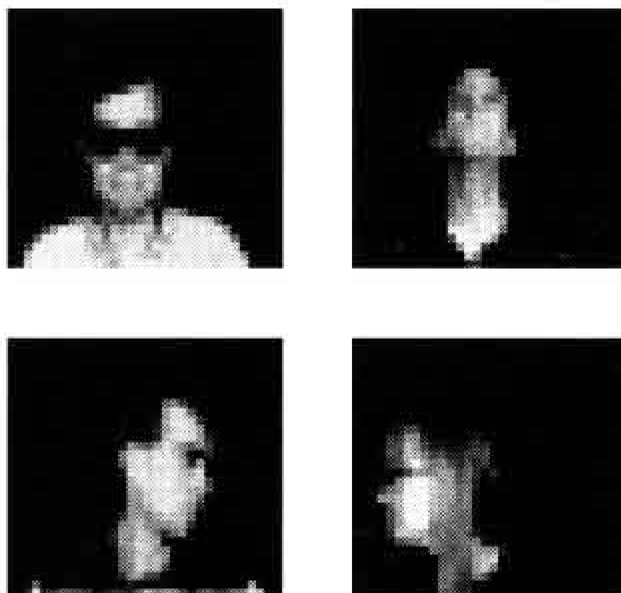
太阳镜网络中连向一个中央神经元的输入权重（比如强度）。

考虑到设置权重并未使用任何人类知识告知太阳镜的一般颜色和位置，这个幻影的出现给人的印象相当深刻，人类提供的信息只有一系列训练图像，每个图像都有个简单的“是”或“否”表明其中是否有太阳镜。太阳镜的幻影从学习阶段反复调整的权重中自动浮现出来。

另外，图片的其他部分也有许多明显的强权重，从理论上讲，这些权重不应该对太阳镜决策有影响。我们怎么才能解释这些没有意义、明显是随机的连接呢？在这里，我们遇到了人工智能研究人员在过去几十年中学到的最重要的教训之一：看似智能的行为有可能从看似随机的系统中浮现出来。如果我们有能力进入人脑，研究神经元之间连接的强度，其中绝大部分连接都会表现得很随机。然而，当作为集合体行动时，这些连接强度的松散集合产生了人的智能行为！



分类正确



分类不正确

太阳镜网络的结果。
资料来源：汤姆·米切尔（Tom Mitchell），《机器学习》（Machine Learning），麦格劳-希尔出版社（1998年）。感谢授权使用该图像。

## 运用太阳镜网络

既然我们现在使用着一个能输出0到1之间任意值的网络，你也许在想我们该怎么才能得到最终答案——图片中的人有没有戴太阳镜？正确的技巧极其简单：超过0.5的输出被视为“有太阳镜”，低于0.5的输出结果被视为“没有太阳镜”。

要测试我们的太阳镜网络，我进行了一次实验。面部数据库包含约600张图片，我将400张图片用于网络学习，然后用剩余的200张图片测试了网络的效果。在这个实验中，太阳镜网络的最终精确度在85%左右。换言之，在太阳镜网络从未见过的图片中，当询问约85%的图片“里面的人戴太阳镜了吗”这个问题时，太阳镜网络都能给出正确答案。上图显示了一些分类正确及分类不正确的图片。检验一个图形识别算法在哪些情况下会失效总是很吸引人，这个神经网络也不例外。上图右侧没有被正确分类的图片中有一两张非常难识别，甚至连人都可能分辨不出。不过，至少有一张图片（右侧图中的左上方图）对人来说非常好辨认——这个男人直视镜头，很明显也戴了太阳镜。这类时不时出现的神秘失败案例在图形识别任务中很常见。

当然，顶尖神经网络在这个问题上的精确度能远高于85%。这里的重点是使用一个简单网络，理解涉及的主要思想。

## 图形识别：过去、现在和未来

我之前提到过，图形识别是更大的领域人工智能（AI）的一个方面。图形识别处理高度变化的输入数据，如音频、照片和视频；AI处理的任务更多元，包括计算机国际象棋、在线聊天机器人和人形机器人。

人工智能是突然从无到有兴盛起来的：在1956年于达特茅斯大学举行的一次会议上，一个由10名科学家组成的小组从根本上创立了这一领域，让“人工智能”这个短语第一次流行开来。在这次大会创建提案——大会组织者将其发给了洛克菲勒基金会——的豪迈文字中，他们的讨论“将在一个推测的基础上进行，这个推测是学习的每个方面或智能的任何其他特性都能从根本上加以精确描述，进而创造一台能模拟它的机器。”

达特茅斯大会给出了很多承诺，但兑现甚少。而每当研究人员确信迈向真正“智能”机器的关键突破就在眼前时，他们的宏大希望总是会被原型机产生的机械行为所打碎。即便神经网络的发展对此帮助也很小：在数波有前景的活动之后，科学家们也用尽了对付这堵机械行为之墙的方法。

不过，人工智能肯定在慢慢地破解那些思想进程集合，这些思想进程可能被定义为只有人类独有。数年来，许多人相信人类国际象棋冠军的直觉和洞见能打败任何计算机程序。计算机程序必须依赖于预设好的规则集，而非直觉。这种看法对人工智能而言本身已摇摇欲坠，并在1997年被彻底去除，IBM的深蓝计算机在这一年打败了世界冠军盖瑞·卡斯帕罗夫（Garry Kasparov）。

同时，人工智能的成功故事也慢慢渗透到普通人的生活中去。通过语音识别来服务消费者的自动电话系统已经很常见。视频游戏中计算机控制的对手开始表现出类似于人的战略，甚至还有个性特点和弱点。亚马逊和Netflix等在线服务开始基于自动推导的个人喜好推荐商品，而结果经常令人极其满意。

的确，我们对这些任务的认知已经从根本上被人工智能的发展改变了。想想这个问题，在1990年肯定需要人类的智能输入来规划一次多地停留的飞机旅行路线，而且人类还会因为自己的专业技能获得报酬。1990年，好的人类旅行中介在寻找方便、低成本的路线上会产生

巨大不同。不过，到2010年，计算机在执行这项任务上就比人类好上很多了。计算机究竟如何达到这一点本身就是个有趣的故事，因为它们在规划路线时的确用到了数个迷人算法。但更重要的是这些系统对我们在这项任务的认知上的影响。我会说，到2010年，大多数人都会认为规划路线的任务是纯粹机械化的行为——和20年前的认知形成鲜明对比。

任务的逐渐转变——从明显是直觉性的任务到显然是机械化的任务——还在继续。普遍意义上以及图形识别中特殊意义上的人工智能正慢慢扩展它们的边界，提升它们的效能。本章描述的算法——最近邻分类器、决策树和神经网络——能被运用于无数实际问题中。这包括纠正手机虚拟键盘上胖手指（fat-fingered）文本输入，从一份复杂的检验结果中帮助诊断病人疾病，在自动收费亭识别汽车牌照，以及决定向某个计算机用户展示什么广告——这里只列举几个。因此，这些算法是图形识别系统的一些基础构件。不管你是否认为它们是真正的“智能”，你都将在未来数年中看到更多这些算法。

# 第七章　数据压缩——有益无害

> 爱玛很快活。如果不是从卧室里传来埃尔顿太太的声音，从而妨碍了她，使她急匆匆地真诚地紧紧握住她的手以表达最诚挚的祝愿和深厚的感情，她真想立刻就表示有话要讲。
>
> ——简·奥斯汀，《爱玛》注

我们对压缩实物的概念都很熟悉：当你试图将许多衣服放入一个小提箱中时，你可以压一压衣服，让它们小到能被小提箱容纳，即便衣服的正常体积会超出小提箱。你压缩了衣服。之后，再从小提箱中拿出衣服时，你就能解压它们，并以它们的原始大小和形状再度穿着（希望能如此）。

令人惊讶的是，我们也有可能对信息做同样的事：计算机文件和其他种类的数据，通常能被压缩成更小的体积，以方便存储或传输。然后，它们会被解压并以原始形式被使用。

绝大多数人的计算机上有足够的磁盘空间，无须为压缩自己的文件烦心。因此，人们倾向于认为，压缩对绝大多数人没有影响。但这种印象是错误的：事实上，计算机系统背后经常用到压缩。比如，许多通过互联网发送的消息都经过了压缩，用户甚至不知道这一点；几乎所有软件都是以压缩格式被下载——这意味着你下载和转移文件的速度，要比不压缩时快数倍。甚至当你对着电话讲话时，你的声音也

经过了压缩：如果电话公司能在传输语音数据前进行压缩，它们就能对自己的资源实现超高利用率。

压缩也以更明显的方式得到了运用。流行的ZIP文件格式运用的精巧压缩算法，将在本章得到介绍。你很有可能也对涉及压缩数字视频的得失很熟悉：高质量视频的文件体积，要比相同视频的低质量版本大很多。

## 无损压缩：终极免费午餐

计算机使用两种截然不同的压缩：无损压缩和有损压缩，这点很重要。无损压缩是真正的免费午餐，对你有益无害。一种无损压缩算法能将一个数据文件压缩为其原始体积的一小部分，然后将其解压为和之前一样的东西。相反，有损压缩会导致解压后的原始文件发生一些小改变。我们稍后再讨论有损压缩，现在让我们专注于无损压缩吧。举个无损压缩的例子，假设原始文件包含本书文本。那么你在压缩前和压缩后得到的文件版本包含完全相同的文本——不会有一个字、空格或标点符号不同。在我们为免费午餐感到欣喜若狂之前，我要加一个重要警告：无损压缩算法并不能为所有文件都节省大量空间。但一个好的压缩算法能为特定大类的文件节省大量空间。

那我们怎么才能享受免费午餐呢？你怎么才能在不破坏的情况下，让一块数据或信息比起实际“真实”体积更小，并在之后完美地重构一切东西呢？事实上，人类一直都在这么做，只不过从未想到过罢了。想一下你的每周日程。为简单起见，让我们假设你每天工作8小时，每周工作5天，你用一小时的区间来划分日程。因此，5个工作日的每天都有8个可能的区间，每周一共有40个区间。然后，将你一周的日程传输给其他人，你必须传输40份信息。但如果有人打电话让你安排下周的一个会议，你会通过列出40份分开的信息来描述自己什么时

候有空吗？当然不会！最有可能的情况是，你会说些“周一和周二全满，周四和周五下午1点到3点有预约，其余时间有空”之类的话。这就是一个无损数据压缩的例子！和你谈话的人能完全重构出你下周所有40个时段的空闲情况，但你却无须详细列出它们。

你也许会想这种“压缩”是取巧，因为它取决于这一事实：你的日程安排中的绝大多数区块都相同。特别是，周一和周二全天都有预约，因此你能非常快速地描述它们，这周剩下的时间里，除了两个时间段以外都有空，这也很容易描述。这的确是个非常简单的例子。不管怎样，计算机中的数据压缩也是按照这一方法运行的：基本思想是发现数据中彼此相同的部分，并运用某种把戏更高效地描述这些部分。

这在数据包含重复片段时尤其简单。比如，你很有可能想出一个压缩下列数据的好方法：

AAAAAAAAAAAAAAAAAAAAABCBCBCBCBCBCBCBCBCBCAAAAAADEFDEFDEF

如果乍一看还不明显，思考一下你会如何通过电话向某人口述这份数据。和说“A、A、A、A、……、D、E、F”不同的是，我肯定你更有可能会说“21个A，然后是10个BC，接着是6个A，最后是3个DEF”。再比如，要很快地在一张纸上记下这份数据，你可能会写“21A、10BC、6A、3DEF”。在这个例子里，你将这个包含56个字母的原始数据压缩成了只有16个字母的字符串。这不到原体积的三分之一——不错！计算机科学家将这种特别的把戏称为行程长度编码（run–length encoding），因为它将重复的“行程”和行程的“长度”编码在了一起。

不幸的是，行程长度编码只在压缩非常特殊的数据种类上有用。它在实际中也有运用，但大部分时候只和其他压缩算法结合起来使

用。比如，传真机就将行程长度编码和另一种被称为霍夫曼编码（Huffman coding）的技术结合起来使用，我们稍后将谈到霍夫曼编码。行程长度编码的主要问题是，数据中的重复片段必须相邻——换句话说，重复部分间不能有其他数据。使用行程长度编码压缩ABABAB很容易（只是3AB），但用相同的把戏压缩ABXABYAB就行不通了。

你也许明白为什么传真机能利用行程长度编码。从定义上来说，传真是黑白文件，文件会被转换成许多个点，每个点都是非黑即白。当你按顺序阅读这些点（从左到右，从上到下），你会遇到大段白点（背景）以及小段黑点（前景文本或笔迹）。这让使用行程长度编码变得非常有效。但正如之前所提到的，只有特定类型的数据具有这一特点。

于是，计算机科学家们发明了一系列更成熟的把戏，这些把戏使用的基本思想都相同（寻找重复并高效地描述它们），但即便重复部分不相邻也效果良好。在这里，我们只会研究其中的两种把戏：同前把戏（same-as-earlier trick）和更短符号把戏（shorter-symbol trick）。你只需要这两个把戏就能生成ZIP文件，而ZIP文件格式是个人电脑上压缩文件最流行的格式。因此，一旦你理解了这两个把戏背后的基本思想，你也就理解了计算机在大部分时间里是如何运用压缩的。

## 同前把戏

想象这就是你要处理的可怕任务，通过电话向某人口述如下数据：

VJGDNQMYLH-KW-VJGDNQMYLH-ADXSGF-OVJGDNQMYLH-ADXSGF-VJGDNQMYLH-EW-ADXSGF

这里有63个字母需要沟通（顺便说一句，我们忽略了破折号，加入它们只是为了让数据更容易阅读）。和每次一个字母地口述全部63个字母相比，我们有更好的办法吗？第一步也许是去识别数据中大量的重复部分。事实上，大多数被破折号分开的“块”都至少重复了一次。因此，在口述这份数据时，你可以通过“这部分和我之前告诉你的某个部分一样”节省大量力气。更精确点讲，你要讲是多久前说的，还要讲重复的部分有多长——也许是“往回数27个字母，然后复制从那一点开始往下的8个字母”。

让我们来看看这一策略在现实中效果如何。最开始的12个字母没有重复部分，你没有其他选择，只能按字母逐个口述：“V、J、G、D、N、Q、M、Y、L、H、K、W”。但接下来的10个字母和之前的一些字母相同，因此你可以说“back 12，copy 10”（数回12个字母，抄到第10个字母）。再下面7个字母是新出现的，按字母逐个口述：“A、D、X、S、G、F、O”。但之后的16个字母是个大的重复部分，因此你可以说“back 17，copy 16”（数回17个字母，抄到第16个字母）。接下来的10个字母也是之前的重复部分，因此“back 16， copy 10” （数回16个字母，抄到第10个字母）就可以了。再接下来的两个字母没有重复，需要逐个口述为“E、W”。最后的6个字母是之前的重复，可以通过“back 18，copy 6”（数回18个字母，抄到第6个字母）沟通。

让我们尝试总结一下这个压缩算法。我们用缩写b代替“back”，c代替“copy”。因此一个如“back 18，copy 6”（数回18个字母，抄到第6个字母）的返回抄写指令简写为b18c6。因此上面的口述指令可以被总结成：

VJGDNQMYLH-KW-b12c10-ADXSGF-O-b17c16-b16c10-EW-b18c6

这个字符串只包含44个字母。原始字符串有63个字母，我们节省了19个字母，接近原始字符串长度的1/3。

同前把戏中还有一个有趣的窍门。你会如何使用相同的把戏压缩FG-FG-FG-FG-FG-FG-FG-FG这条消息？（和之前一样，破折号不是消息的一部分，只是为增强可读性而添加。）消息中的FG有8处重复，因此我们可以单个口述前4个，然后使用如下的返回抄写指令：FG-FG-FG-FG-b8c8。这会节省不少字母，但我们可以做得更好。这需要一个第一眼看起来可能显得荒谬的返回抄写指令："back 2， copy 14"（数回2个字母，抄到第14个字母），或简写为b2c14。压缩的消息就成了FG-b2c14。在只有2个字母可供抄写的情况下，怎么理解抄到第14个字母呢？事实上，只要你从重新生成的消息中而非压缩的消息中抄写，就不会有任何问题。让我们一步步地来做。在口述了最开始的2个字母后，我们有了FG。然后我们收到b2c14指令，于是我们数回2个字母并开始抄写。因为只有2个字母（FG），让我们抄写这2个字母：当把抄写的字母加到我们已有的字母后，结果是FG-FG。但现在多了2个字母！照样抄写这些字母，在将它们添加到已有的重新生成的消息上之后，你得到了FG-FG-FG。和前一次一样，又多出2个字母，于是你又能多抄写2个字母。依此类推，直到你抄写了所要求的字母数（在这个例子中就是14个）。要检验自己是否理解了这一点，看看你能否得到这条压缩消息的解压版：Ab1c250[注]。

## 更短符号把戏

要理解名为"更短符号把戏"的压缩把戏，我们要更深入探究计算机存储消息的方式。你之前可能听说过，计算机并不真的存储a、b、c这样的字母。所有东西都以数字存储，然后根据一些固定表格翻译为字母。（这种在字母和数字之间转换的技术在校验和的讨论中有提到。）比如，我们可以同意"a"由27代表，"b"由28代表，"c"由29代表。那么字符串"abc"就可以以"272829"的形式存储在计算机中，而在屏幕上显示或打印在纸上之前，这些数字又能轻易翻译回"abc"。

下面的表格给出了一个完整列表，列出了计算机也许想存储的100个符号以及每个符号对应的两位数代码。顺便说一下，这个特别的两位数代码集并没有在任何现实计算机系统使用，但与现实生活中使用的代码相当相似。两者的主要区别是，计算机并不使用人类使用的十进制数系统。相反，也许你知道，计算机使用一个不同的数位系统，这个系统被称为二进制系统。不过，这些细节对我们来说并不重要。更短符号压缩把戏同时对十进制和二进制数系统奏效，因此我们假装计算机使用十进制，以使接下来的解释更容易。

仔细看看这张符号表。注意表的第一项给出了字与字之间空格的数字代码“00”。接下来是从A（“01”）到Z（“26”）的大写字母，以及从a（“27”）到z（“52”）的小写字母。再接下来是标点符号，最后一栏中收录了一些书写非英语单词用到的字符，从á（“80”）到ù（“99”）。

那么计算机该如何使用这些两位数代码存储“Meet your fiancé there”（去那见你的未婚夫）这句话呢？很简单：只要将每个字符翻译成对应的数字代码并串联在一起：

M e e t y o u r f i a n c é t h e r e .

13 31 31 46 00 51 41 47 44 00 32 35 27 40 29 82 00 46 34 31 44 31 66

在计算机中，数字对之间是没有间隔的，认识到这一点很重要。因此，这条消息实际上被存储为一个46位数的持续字符串：“1331314600514147440032352740298200463431443166”。当然，人类解读这个字符串有点难，但对计算机来说却轻而易举。在将数字翻译成字符显示在屏幕上之前，计算机能轻易将数字分成对。关键是在如何分开数字代码上没有歧义，因为每个代码都只是用两个数字。事实上，这也是A用“01”而不是“1”代表的原因。同理，B是“02”而不是“2”，一直到字母I（“09”而不是“9”）。如果我们选择让A=“1”，

B="2"，依此类推，根本就不可能清楚地翻译消息。比如，消息"1123"可以拆成"1 1 23"（翻译为AAW），或"11 2 3"（KBC）或"1 1 2 3"（AABC）。因此，请记住这一重要思想：数字码和字符之间的翻译必须清楚无异议，即便代码在没有空格的情况下彼此相邻地存储。我们马上就要为这个问题感到头痛！

回到更短符号把戏。和本书中描述的许多本该是技术性的思想一样，人类也一直在运用更短符号把戏，用时甚至想都没想过。更短符号把戏的基本思想是，如果你使用某样东西足够多次，给它起个简短缩写是很值得的。所有人都知道"USA"是"United States of America"（美利坚合众国）的缩写——我们所有人每次在输入或说出这个由3个字母组成的代码"USA"，而非其代表的由24个字母组成的完整短语时都节省了很多力气。你知道"The sky is blue in color"（天空很蓝）的缩写吗？这个短句恰好也由24个字母组成。当然不知道！但为什么呢？"United States of America"和"The sky is blue in color"之间有什么区别？关键区别在于，其中一个短语的使用频率要比另一个大得多，而通过缩写一个经常使用的短语而非一个极少使用的短语，我们可以节省大量力气。

<table>
<tr><td>space</td><td>00</td><td>T</td><td>20</td><td>n</td><td>40</td><td>(</td><td>60</td><td>á</td><td>80</td></tr>
<tr><td>A</td><td>01</td><td>U</td><td>21</td><td>o</td><td>41</td><td>)</td><td>61</td><td>à</td><td>81</td></tr>
<tr><td>B</td><td>02</td><td>V</td><td>22</td><td>p</td><td>42</td><td>*</td><td>62</td><td>é</td><td>82</td></tr>
<tr><td>C</td><td>03</td><td>W</td><td>23</td><td>q</td><td>43</td><td>+</td><td>63</td><td>è</td><td>83</td></tr>
<tr><td>D</td><td>04</td><td>X</td><td>24</td><td>r</td><td>44</td><td>,</td><td>64</td><td>í</td><td>84</td></tr>
<tr><td>E</td><td>05</td><td>Y</td><td>25</td><td>s</td><td>45</td><td>-</td><td>65</td><td>ì</td><td>85</td></tr>
<tr><td>F</td><td>06</td><td>Z</td><td>26</td><td>t</td><td>46</td><td>.</td><td>66</td><td>ó</td><td>86</td></tr>
<tr><td>G</td><td>07</td><td>a</td><td>27</td><td>u</td><td>47</td><td>/</td><td>67</td><td>ò</td><td>87</td></tr>
<tr><td>H</td><td>08</td><td>b</td><td>28</td><td>v</td><td>48</td><td>:</td><td>68</td><td>ú</td><td>88</td></tr>
<tr><td>I</td><td>09</td><td>c</td><td>29</td><td>w</td><td>49</td><td>;</td><td>69</td><td>ù</td><td>89</td></tr>
<tr><td>J</td><td>10</td><td>d</td><td>30</td><td>x</td><td>50</td><td><</td><td>70</td><td>Á</td><td>90</td></tr>
<tr><td>K</td><td>11</td><td>e</td><td>31</td><td>y</td><td>51</td><td>=</td><td>71</td><td>À</td><td>91</td></tr>
<tr><td>L</td><td>12</td><td>f</td><td>32</td><td>z</td><td>52</td><td>></td><td>72</td><td>É</td><td>92</td></tr>
<tr><td>M</td><td>13</td><td>g</td><td>33</td><td>!</td><td>53</td><td>?</td><td>73</td><td>È</td><td>93</td></tr>
<tr><td>N</td><td>14</td><td>h</td><td>34</td><td>"</td><td>54</td><td>{</td><td>74</td><td>Í</td><td>94</td></tr>
<tr><td>O</td><td>15</td><td>i</td><td>35</td><td>#</td><td>55</td><td>|</td><td>75</td><td>Ì</td><td>95</td></tr>
<tr><td>P</td><td>16</td><td>j</td><td>36</td><td>$</td><td>56</td><td>}</td><td>76</td><td>Ó</td><td>96</td></tr>
<tr><td>Q</td><td>17</td><td>k</td><td>37</td><td>%</td><td>57</td><td>-</td><td>77</td><td>Ò</td><td>97</td></tr>
<tr><td>R</td><td>18</td><td>l</td><td>38</td><td>&</td><td>58</td><td>Ø</td><td>78</td><td>Ú</td><td>98</td></tr>
<tr><td>S</td><td>19</td><td>m</td><td>39</td><td>'</td><td>59</td><td>ø</td><td>79</td><td>Ù</td><td>99</td></tr>
</table>

计算机用于存储符号的数字码

让我们尝试将这个想法应用到上一页的代码系统中去。我们已经知道，通过使用经常用到的东西的缩写，我们节省的力气能达到最大。字母“e”和“t”在英语中使用得最频繁，让我们尝试用更短的代码来代替这两个字母。现在，“e”是31，“i”是46——每个字母都需要两个数字代表。将两个数字减成一个数字呢？假设现在“e”由8代表，“t”由9代表。这个主意太好了！记得我们之前是如何编码短句“Meet your fiancé there”（去那见你的未婚夫）的吧，当时一共用了46个数字。现在我们可以只使用40个数字：

M e e t y o u r f i a n c é t h e r e .

138 8 9 005141474400323527402982009 348 448 66

不幸的是，这一计划有个致命缺陷。计算机并不存储单个字母间的空格。因此编码不会像“13 8 8 9 00 51 . . . 44 8 66”，而是和“138890051. . . 44866”一样。你发现问题了没有？集中看前5个数字，也就是13889。注意代码13代表“M”，8代表“e”，9代表“t”，因此数字13889的解码方式之一是将其拆成13-8-8-9，得到单词“Meet”。但88代表重读符号“ú”，因此数字13889也可能拆分为13-88-9，即“Mút”。事实上，情况还可以更糟，因为89代表另一个略为不同的重读符号“ù”，因此13889的另一种可能拆分是13-8-89，代表“Meù”。根本没办法分辨这三种可能的翻译中哪种正确。

这是一场灾难！我们使用更短的代码代表字母“e”和“t”的精巧计划，导致了一个根本不奏效的编码系统。幸运的是，再多使用一种把戏就能弥补。真正的问题是当我们看到数字8或9时，我们无从分辨它是一个一位数代码（不管是“e”还是“t”）的一部分，还是一个以8或9开头的两位数代码（如重读符号“á”和“è”等）的一部分。要解决这个问题，我们必须有所牺牲：一些代码实际上会变得更长。以8或9开头的清楚的两位数代码会变成三位数代码，但并不会以8或9开头。下文的图表展示了一种达成这一目的的特殊方法。一些标点符号也受到了影

响，但最终情况良好：任何以7开头的数字都是三位数代码，任何以8或9开始的数字都是一位数代码，任何以0、1、2、3、4、5或6开头的数字都是和前面一样的两位数代码。因此，现在拆分数字13889的方法只有一种（13-8-8-9，代表“Meet”）——这也是用于其他任何正确编码的数字字符串。所有模糊性都被消除了，原始消息的新编码如下：

M e e t y o u r f i a n c é t h e r e .

138 8 9 0051414744003235274029782009 348 448 66

原始编码使用了46个数字，这次只使用了41个数字。这看起来也许只是节省了一点点，但如果是一条更长的信息，节省的长度就会非常明显。比如，本书文本（只有文字，没有图像）要求将近500 KB——50万字符——的存储空间。但在使用了刚刚描述的两种把戏压缩后，本书文本的体积减少到只有160 KB，不到原体积的三分之一。

| space | 00 | T | 20 | n | 40 | ( | 60 | á | 780 |
|---|---|---|---|---|---|---|---|---|---|
| A | 01 | U | 21 | o | 41 | ) | 61 | à | 781 |
| B | 02 | V | 22 | p | 42 | * | 62 | é | 782 |
| C | 03 | W | 23 | q | 43 | + | 63 | è | 783 |
| D | 04 | X | 24 | r | 44 | , | 64 | í | 784 |
| E | 05 | Y | 25 | s | 45 | - | 65 | ì | 785 |
| F | 06 | Z | 26 | **t** | **9** | . | 66 | ó | 786 |
| G | 07 | a | 27 | u | 47 | / | 67 | ò | 787 |
| H | 08 | b | 28 | v | 48 | : | 68 | ú | 788 |
| I | 09 | c | 29 | w | 49 | ; | 69 | ù | 789 |
| J | 10 | d | 30 | x | 50 | < | 770 | Á | 790 |
| K | 11 | **e** | **8** | y | 51 | = | 771 | À | 791 |
| L | 12 | f | 32 | z | 52 | > | 772 | É | 792 |
| M | 13 | g | 33 | ! | 53 | ? | 773 | È | 793 |
| N | 14 | h | 34 | " | 54 | { | 774 | Í | 794 |
| O | 15 | i | 35 | # | 55 | \| | 775 | Ì | 795 |
| P | 16 | j | 36 | $ | 56 | } | 776 | Ó | 796 |
| Q | 17 | k | 37 | % | 57 | – | 777 | Ò | 797 |
| R | 18 | l | 38 | & | 58 | Ø | 778 | Ú | 798 |
| S | 19 | m | 39 | ’ | 59 | ø | 779 | Ù | 799 |

使用更短符号把戏的数字码。与上一个表格的不同以黑体显示，两个常用字母的代码被缩短了，代价是用更大的数字增长了不常用符号的代码。结果是大部分消息的总长度缩短。

## 总结：免费午餐来自何方？

现在，我们理解了创建计算机标准压缩ZIP文件背后的所有重要概念。下面是具体实施步骤：

**步骤一：**

计算机使用同前把戏传输未经压缩的原文件，让文件中绝大多数重复数据由短得多的指令取代，这些指令会返回并拷贝其他地方的数据。

**步骤二：**

计算机会检查传输后的文件，选出经常出现的符号。比如，如果原文件以英语为书写语言，那么计算机就很有可能会发现“e”和“t”是最常出现的两个符号。随后，计算机会创建一张如下页的表格，用短数字码代表经常用到的符号，用更长的数字码代表极少用到的符号。

**步骤三：**

计算机会通过直接将文件翻译为步骤二中的数字码来再次传输文件。

步骤二中计算出的数字码表也会存储在ZIP文件中，否则在后面不可能解码（并解压）ZIP文件。注意，不同的未压缩文件会得到不同的数字码表。事实上，在真正的ZIP文件中，原文件被分成了小块，每个块都有不同的数字码表。计算机能自动高效地完成所有这些步骤，实现多种文件的优质压缩。

## 有损压缩：不是免费午餐，但也是一笔好买卖

到目前为止，我们一直都在讨论无损压缩，因为你能将压缩过的文件重新组建成一开始使用的原文件，连一个字母或一个标点符号都

没有改变。相反，有时候使用有损压缩要有用得多。有损压缩能让你将一个压缩过的文件重新组建成一个和原文件非常类似，但并不完全和原文件相同的文件。比如，有损压缩经常用于包含图片或音频数据的文件：只要照片在人眼中看起来一样，在计算机上存储照片的文件是否和在相机中存储照片的文件相同其实并不重要。同样的道理也适用于音频数据：只要歌曲在人耳中听起来一样，在数字音乐播放器上存储歌曲的文件是否和在CD中存储歌曲的文件相同也并不重要。

事实上，有时候有损压缩会以更为极端的方式使用。我们都看过互联网上的低质量视频和图片，里面的画质模糊，音质糟糕。这是过度使用有损压缩的结果，目的是让视频或图像文件体积变得非常小。我并不是要说视频看起来要和原视频一样，但至少也要可以看清。通过调整压缩的“损失率”，网站操作人员可以在看起来和听起来几近完美的高质量大文件，以及有着明显缺陷但传输带宽要求很少的低质量文件之间进行选择。你也可能在数码相机上做过同样的事，通常你能选择图像和视频质量的不同设置。如果你选择高质量设置，你能在相机上存储的照片或视频数，就会比选择低质量设置时少很多。这是因为高质量媒体文件要比低质量媒体文件占据更多空间。而这一切都只需通过调整压缩“损失率”来完成。接下来，我们将探究几种进行这一调整的把戏。

## 抛弃把戏

有损压缩的一个简单且有效的方法是直接抛弃一些数据。让我们来研究一下“抛弃”把戏（leave-it-out trick）如何在黑白照片上运用。我们首先要了解一点黑白照片如何在计算机上存储的知识。照片由大量小点组成，这些小点被称为“像素”。每个像素只有一种颜色，这种颜色可以是黑，也可以是白，或黑与白之间的任意灰度。当然，我们

通常不会意识到这些像素，因为它们太小了，但如果你离显示器或电视屏足够近，就能看到单个像素。

在计算机中存储的黑白照片里，每个可能的像素颜色都由一个数字代表。比如，让我们假设越大的数字代表越白的颜色，100是最大的数字。因此100代表白色，0代表黑色，50代表中等灰度，90代表浅灰等等。像素按照矩形方阵排列，其中每个像素都代表图片中一些非常小的部分的颜色。方阵的行列总数就是图像的“解析度”。比如，许多高清电视的解析度是1 920×1 080，这意味着图像有1 920列和1 080行像素。像素总数就是1 920乘以1 080，也就是逾两百万像素！数字相机也使用相同的术语。“megapixel”只是个用来表示百万像素的花哨名字。因此，一台500万像素的相机有足够多行和足够多列的像素，当你将行数和列数相乘后,得到的数就会超过500万。当照片存储在计算机中时，它就是一个数字列表，每个像素都由数字代表。

下图中左上方的带角楼的房屋照片,比高清电视的解析度低很多：只有320×240。不过，其像素数目仍然相当大（320×240 = 76 800），存储这张照片的文件,在未压缩形式下占用了超过230 KB的存储空间。1 KB约相当于1 000个文本字符——差不多是一段电子邮件的体积。因此，左上方的图在作为文件存储时，需要占用的空间约等于200条短电子邮件消息。

我们可以用下列非常简单的技术来压缩这个文件：每两行或每两列像素就忽略或“抛弃”一行或一列。抛弃把戏就是这么简单！在这个例子中，结果是得到了一张解析度更小的照片，这张解析度为160×120的新照片在图中原照片下面显示。新文件的体积只有原文件的四分之一（约57 KB）。这是因为新图片的像素只有原图片像素的四分之一多——我们同时去除了原图片宽和高各一半的像素。相当于原图片的体积减小了一半——一次是水平方向，一次是垂直方向——结果就是最终体积只有原体积的1/4。

我们还可以再用一次抛弃把戏。将新的160×120像素图片每两行或每两列像素抛弃一行或一列，得到另一张80×60的新图片，结果显示在下图的左下方。这张图片的体积又缩小了3/4，其最终体积只有14 KB。相当于原始文件体积的6%，这一压缩比率令人印象非常深刻。



320 × 240像素

[230KB]

压缩



160 × 120像素

[57KB]

解压



解压自160 × 120像素

[57KB]



80 × 60像素

[14KB]

解压



解压自80 × 60像素

[14KB]

运用抛弃把戏的压缩，左边显示的是原始图片，以及两张该图片的缩减版。每张缩减后的图片，都通过抛弃前一张图片一般的行和列计算得出。右边是将缩减后的图片解压到原始图片同样大小时的解压效果图。重构并不完美，重构图和原图有明显区别。

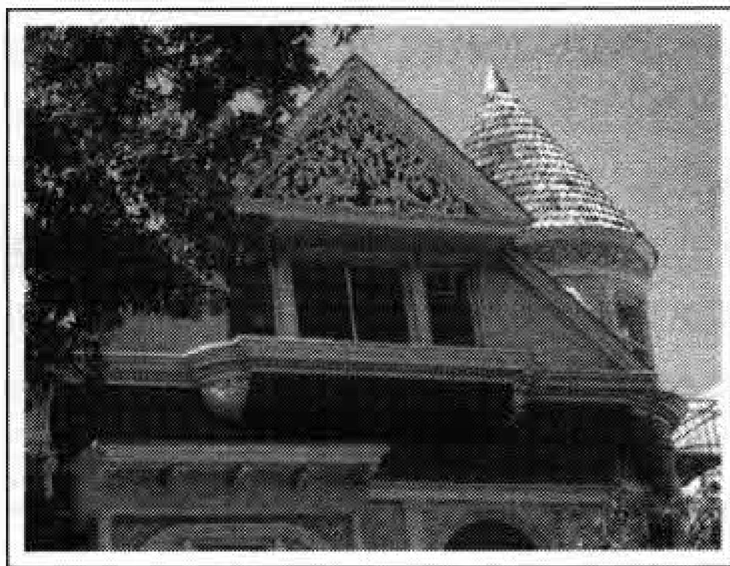
但请记住，我们在使用有损压缩，并没有免费午餐。午餐很便宜，但我们必须付钱。当我们将压缩过的文件之一解压到原图大小时，看看发生了什么。因为其中一些像素行和列被删除，计算机必须推测这些消失像素的颜色是什么。最简单的可能推测是给任一消失像素以其相邻像素之一的颜色。选择任一相邻像素都可以，但在这个例子里，计算机选择了消失像素正上方和左边的像素。

这一解压机制的结果显示在图右侧。能看到大部分视觉特征得到了保留，但在画质和细节上有一些明显损失，特别是在树、角楼房顶以及房屋山形墙的格子通气孔等复杂区域。另外，特别是在解压自80×60图片的版本中，你能在房屋屋顶的斜线等区域看到一些相当令人不悦的锯齿。这些现象被我们称为“压缩缺陷”（compression artifact）：不仅仅是细节的损失，而且有损压缩的某种方法会在接下来的解压中引入明显的新特征。

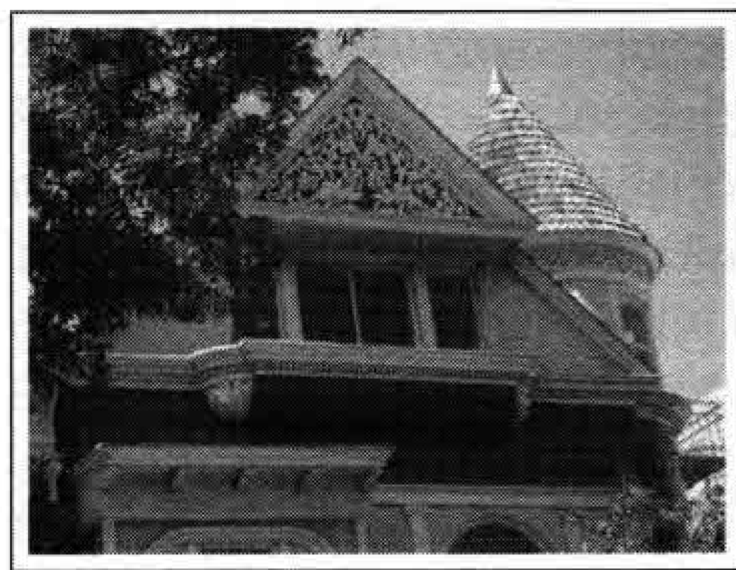
尽管抛弃把戏在理解有损压缩的基本思想上很有用，但在这里描述的简单抛弃把戏却极少用到。计算机的确会“抛弃”信息以实现有损压缩，但它们对抛弃哪些信息却要小心得多。这方面的常见例子之一，就是JPEG图像压缩格式。JPEG是一种精心设计的图像压缩技术，它在效果上要比每两行（列）抛弃一行（列）的抛弃把戏好很多。仔细看看下面的图，将图片的质量和大小与上图相比。最上方的JPEG图片大小为35 KB，但几乎和原始图片相同。通过抛弃更多信息并继续使用JPEG格式，我们得到了中间大小为19 KB的图。尽管房屋的格子通气孔有些模糊和细节上的损失，但中间图片的质量依然很棒。但如果压缩得太厉害，JPEG格式也会有压缩缺陷：最底部的JPEG图被压缩到只有12 KB，你能看到天空中出现了一些块状效果，在房顶斜线右边的天空也出现了一些令人不快的斑点。

尽管JPEG抛弃战略的细节太过技术化，在这里不能完整描述，但这项技术的基本原理还算好说。JPEG首先将整张图片划分为8×8像素

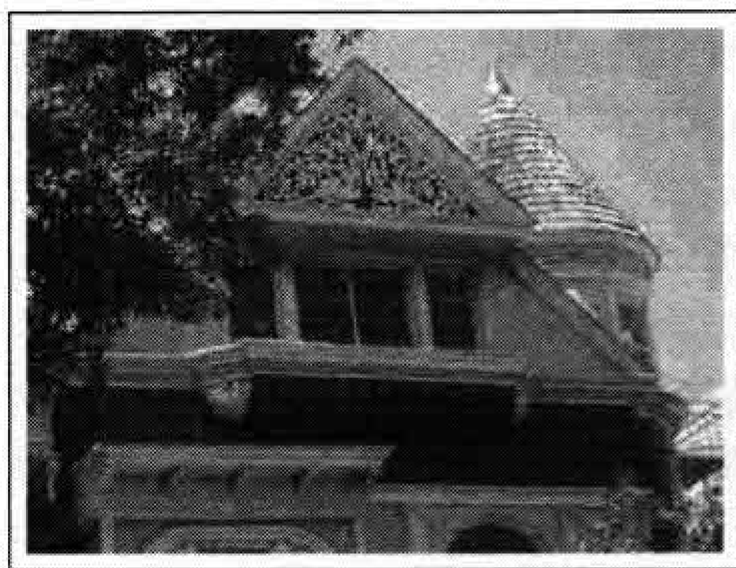
的小方块，每个方块都会被单独压缩。注意，在没有压缩的情况下，每个方块代表8×8=64个数字。（我们假设这是张黑白照片——如果它是彩色图片，那么就会有三种不同的颜色，数字会是原来的三倍，但在这里我们不要担心这一细节。）如果方块恰好只有一种颜色，整个方块就能由一个数字代表，而计算机就能“抛弃”63个数字。如果方块大部分由一种颜色组成，只有少数像素的颜色略有不同（也许一片天空的灰度都相同），计算机也可以用单个数字代表方块，让方块得到好的压缩结果，并在稍后解压时只出现少量错误。在上图中的最后一张图中，天空中一些8×8的块，就是采用这种方法压缩，导致出现了一些单色方块。



JPEG[35KB]



JPEG[19KB]



JPEG[12KB]

使用有损压缩机制，更高的压缩得到的画质更低。这里展示了使用三种不同JPEG质量级别压缩同一张照片后得到图片。最上面的图片质量最高，同时要求的存储空间也最多。最下面的图片质量最低，要求的存储空间不到最上面图片的一半，但却有明显的压缩缺陷——特别是在天空和房顶边缘。

如果8×8方块从一种颜色渐变为另一种颜色（比如左边是深灰色，右边是浅灰色），那么64个数字也许能被压缩到只有2个：一个深灰的值和一个浅灰的值。JPEG算法并不一定这么运作，但它使用了相

同的思想：如果一个8×8方块符合一些已知模式如时时色（constant color）或渐变色（smoothly varying color）的组合，那么其大部分信息就可以被抛弃，只需存储每个模式的级别或量即可。

JPEG格式在图片上效果良好，但对音频和音乐文件呢？这些文件也能使用有损压缩去压缩，而且使用了相同的基本思想：抛弃对成品影响很小的信息。MP3和AAC等流行音乐压缩格式通常使用和JPEG一样的高级方法。音频也会被划分成块，每个块都会被单独压缩。在JPEG中，以特定方式变化的块能用少量数字形容。不过，音频压缩格式也能利用与人耳有关的已知事实。特别是，有些种类的声音对人只有很小的影响或没有影响，因此压缩算法能在不降低输出质量的情况下消除这些声音。

## 压缩算法的起源

本章描述的同前把戏——用于ZIP文件中的压缩方法之一——以LZ77算法为计算机科学家所熟知。这一算法由两位以色列计算机科学家亚伯拉罕·伦佩尔（Abraham Lempel）以及雅各布·齐夫（Jacob Ziv）于1977年发表。

不过，要追溯压缩算法的起源，我们需要把科学史向前推30年。我们已经了解了克劳德·香农，那位以其1948年论文创建信息理论领域的贝尔实验室科学家。香农是纠错码故事中（第五章）的两位主要英雄之一，但他以及他于1948年发表的论文在压缩算法的出现上也扮演了重要角色。

这并非偶然。事实上，纠错码和压缩算法是同一枚硬币的两面。两者都来自冗余的想法，我们在第五章详细介绍了冗余的概念。如果一个文件有冗余，它就比必要的长度长。这里重复一个第五章的简单

例子，文件可以使用单词“five”来代替数字“5”。那样的话，像“fivq”这样的错误就能被轻易识别和纠正。因此，纠错码能被视为向消息或文件中添加冗余的原则性方法。

压缩算法正好相反：它们会从消息或文件中移除冗余。很容易想象一个压缩算法会注意到文件中经常使用单词“five”，并用一个更短的符号取代它（甚至有可能是符号“5”），正好反转了纠错码过程。在现实中，压缩和纠错并不会像这样彼此抵消。相反，好的压缩算法会移除低效冗余，而纠错编码会增加另一种更高效的冗余。因此，首先压缩一条信息，再往里面添加一些纠错码非常常见。

再说回香农，他于1948年发明的重要论文除了许多卓越贡献之外，还包含对最早期压缩技术之一的描述。麻省理工学院教授罗伯特·法诺（Robert Fano）大约在同时发明了这一技术，这一方法现在以香农—法诺编码（Shannon-Fano coding）命名。事实上，香农—法诺编码是一种实施更短符号把戏的特殊方法，我们在本章前面描述了更短符号把戏。我们很快就会知道，香农—法诺编码很快就被另一种算法取代，但这一方法非常有效，并存活到了今天，成为ZIP文件格式的可选压缩方法之一。

香农和法诺都意识到，尽管他们的方法都既实用又高效，但却不是最好的算法：香农通过算术证明了肯定有更好的压缩技术存在，但还未找到实现它们的方法。同时，法诺在麻省理工教授一门信息理论的研究生课程，他将实现优化压缩的问题作为该课程学期论文的可选项之一。出人意料的是，法诺的一位学生解决了这一问题，得到了一种针对每个符号取得最佳可能压缩的方法。这名学生就是大卫·霍夫曼，他的技术——现在以霍夫曼编码来命名——则是更短符号把戏的另一个例子。霍夫曼编码仍然是一种基础压缩算法，被广泛用于通信和数据存储系统。

---

1. 译文选自张宇译本。——译者注
2. 原注：答案是字母A重复251次。

# 第八章　数据库——追求一致性的征程

“数据！数据！数据！”他不耐烦地喊道：“没有数据怎么推导。”

——阿瑟·柯南·道尔
《福尔摩斯探案集》之《桐山毛榉案》
（The Adventure of the Copper Beeches）

想象如下神秘仪式：一个人从桌上拿起一沓特别印制的纸（支票簿），在上面写下一些数字，再加上一个带花饰的签名。随后，这个人把这沓纸最上面的一张纸撕下来，放进信封，再用另一张纸（邮票）贴在信封正面。最后，这个人把信封带到外面，走过街道尽头，将信封放入一个大箱子内。

这是任何交账单的人——电话账单、电费账单、信用卡账单等——每月都要进行的仪式。直到21世纪之交时，在线账单支付和在线银行系统才发展起来。和这些系统的简便性相比，之前基于纸张的方法费时费力、极其低效。

哪些技术让这一转变成为可能？最显然的答案是互联网的到来，没有互联网，任何形式的在线通信都不可能。另一项关键技术是公钥加密，这一点我们已经在第四章讨论过。没有公钥加密，机密财务信息就不能通过互联网安全地进行传输。不过，在线交易至少还需要另一种至关重要的技术：数据库。幸好绝大多数计算机用户不用意识到数据库的存在，但基本上所有在线交易都会使用成熟的数据库技术处理。计算机科学家们自从20世纪70年代就开始开发数据库技术。

数据库针对交易处理中的两个主要问题：高效性和可靠性。数据库通过算法实现高效性和可靠性，这些算法允许成千上万的客户在不产生任何冲突或不一致的情况下同时进行交易，也可以在硬盘等计算机组件出现故障的情况下保持数据完整，而硬盘故障通常会导致严重的数据损失。在线银行就是一个标准的需要卓越高效性（同时服务众多客户且不能产生任何错误或不一致）和近乎完美的可靠性的应用例子。为收束我们的讨论，我们会经常返回在线银行的例子。

本章中，我们将了解数据库背后三种美丽的基础思想：预写日志记录（write-ahead logging）、两阶段提交（two-phase commit）和关系数据库（relational database）。这些思想让存储特定种类重要信息的数据库技术占据了绝对的主宰地位。和前面一样，我们将尝试专注于每种思想背后的核心洞见，认识一种让思想奏效的把戏。预写日志记录的基础是“待办事项表把戏”（to-do list trick），我们将首先处理这一把戏。之后，我们会转移到两阶段提交协议，用简单但强大的“预备提交把戏”（prepare-thencommit trick）描述。最后，我们会通过了解有关“虚表把戏”（virtual table trick）的知识，一窥关系数据库的世界。

但在学习这些把戏之前，让我们先尝试消除数据库的神秘感，明白数据库是什么。事实上，即便是在技术性计算机科学文献中，“database”（数据库）这个单词都代表很多不同的事物，因此不可能给它下一个单一的、正确的定义。但绝大多数专家都同意，数据库有别于其他存储信息的方式的关键特征是，数据库中的信息有一个预定义结构。

要理解这里的“结构”意味着什么，让我们首先来看一个非结构化信息的例子：

> 罗西娜35岁，她和26岁的玛特是朋友。静宜37岁，苏迪普31岁。玛特、静宜和苏迪普都是朋友。

这就是Facebook或MySpace等社交网站需要存储的与其成员有关的那类信息。不过，这些社交网站自然不会如此毫无结构地存储信息。下面是相同信息的结构化形式：

| **名字** | **年龄** | **朋友** |
|---|---|---|
| 罗西娜 | 35 | 玛特 |
| 静宜 | 37 | 玛特，苏迪普 |
| 玛特 | 26 | 罗西娜，静宜，苏迪普 |
| 苏迪普 | 31 | 静宜，玛特 |

计算机科学家称这种结构为表（table）。表的每一行都包含和单件事情（在这个例子里就是一个人）有关的信息。表的每一栏都包含一个特定种类的信息，如一个人的年龄或名字。数据库通常由许多表组成，但为简单起见，我们最开始的例子只会使用一个表。

很显然，让人和计算机操纵结构化表格中的数据，要比上个例子中非结构化的自由文本远为高效。但数据库对它们的作用要远比便于使用多得多。

我们进入数据库世界的旅程将从一个新概念开始：一致性（consistency）。我们很快就会发现，数据库从业者痴迷于一致性，而且这么做还有着很好的理由。从简单意义上来说，“一致性”指数据库中的信息并不自相矛盾。如果数据库中有矛盾之处，我们就碰到了数据库管理员最糟糕的噩梦：不一致。不过，一开始不一致是如何产生的呢？想象上表的前两行略为改变：

| **名字** | **年龄** | **朋友** |
|---|---|---|
| 罗西娜 | 35 | 玛特，静宜 |
| 静宜 | 37 | 玛特，苏迪普 |

你能发现问题所在吗？从第一行看，罗西娜是静宜的朋友。但从第二行来看，静宜并没有成为罗西娜的朋友。这违背了朋友的基本概念，只有两个人同时是朋友，才能称彼此为朋友。不得不承认，这个不一致的例子相当温和。想象一个更严肃的例子，假设用“婚姻”的概念取代“朋友”。那么结果会是A和B结了婚，但B又和C结了婚——这种情况在许多国家都违法。

实际上，当新数据加入数据库时，这种不一致很容易避免。计算机非常擅长照章办事，因此让数据库遵循“如果A和B结了婚，那么B必须也和A结婚”的规则会很容易。如果有人试图输入一条违反这条规则的新行，他们会收到一条错误消息，输入也会失败。因此，在简单规则的基础上确保一致性并不需要任何机巧把戏。

但其他种类的不一致需要更为精巧的解决方案。我们将在下一部分研究其中一种不一致情况。

## 事务和待办事项表把戏

事务（Transaction）极有可能是数据库世界中最重要的思想。但要理解它们是什么，以及它们为什么必要，我们需要接受有关计算机的两个事实。你也许非常熟悉第一个事实：计算机程序会崩溃——当一个程序崩溃时，它会丢掉所有正在处理的东西。只有安放在计算机文件系统中的信息会得到保存。我们要知道的第二个事实不太为人所知，但却极其重要：硬盘和闪存条等计算机存储设备一次只能写入少量数据——基本上在500个字符左右。（如果你对技术术语感兴趣，我这里说的是硬盘扇区大小<sector size>，通常一个扇区可以存放512个字节的信息。而闪存的相关度量单位是页大小<page size>，闪存每页能存储成千上万个字节的信息。）作为计算机用户，我们从未注意在一台设备上存储数据时的小容量限制，因为现代设备每秒能执行成千

上万次这种500个字符的写入操作。但磁盘内容每次只能改变数百个字符的现实依旧。

这究竟和数据库有什么关系？它产生了一个极其重要的后果：一般来说，计算机在任何时候都只能更新一行数据库信息。不幸的是，上面那个非常小的简单例子并没有展示这一点。上面整个表只包含不到200个字符，因此在这个特例里，计算机应该能一次更新两行。但通常来说，对于任何一个大小合理的数据库而言，更改两行的确需要两次单独的磁盘操作。

在了解了这些背景后，我们就能触及问题核心了。许多对数据库看似简单的变更要求更改两行或更多。我们现在已经知道，更改两行不能通过一次磁盘操作实现，因此数据库更新会导致两次或更多的磁盘操作。但计算机可能在某个时间崩溃。如果计算机在这样的两次磁盘操作之间崩溃怎么办？计算机会重启，但它会忘掉之前计划执行的操作，有可能一些必要更改从未进行。换言之，数据库可能处于不一致状态！

在这种情况下，崩溃后的不一致问题看起来也许会相当学术化。因此，我们将通过两个例子来研究这一极端重要的问题。让我们从一个更简单的数据库开始：

| 名字 | 朋友 |
|---|---|
| 罗西娜 | 无 |
| 静宜 | 无 |
| 玛特 | 无 |

这个非常单调、令人沮丧的数据库列出了三个孤独的人。现在假设罗西娜和静宜成为朋友，我们要更新数据库来反映这一令人高兴的事件。正如你知道的，这一更新需要同时更改表前两行——而且正如

我们之前讨论的，这通常会要求两次单独的磁盘操作。假设我刚刚更新了第一行。在这次更新后，在计算机有机会执行第二次磁盘操作更新第二行前，数据库的情况如下：

| 名字 | 朋友 |
| --- | --- |
| 罗西娜 | 静宜 |
| 静宜 | 无 |
| 玛特 | 无 |

到目前为止，情况一切良好。现在，数据库程序只需更新第二行，任务就完成了。但等等：如果计算机在数据库程序有机会这么做之前就崩溃了呢？那么在计算机重启后，它不会知道第二行仍需要更新。数据库会像前面一张图一样：罗西娜和静宜成了朋友，但静宜没有和罗西娜成为朋友。这就是让人害怕的不一致性。

我已经提到过，数据库操作员痴迷于一致性，但直到现在，一致性看起来也不怎么像个大事。毕竟，静宜在一个地方被记录和某人成为朋友，在另一个地方的记录里则没有朋友，这真的重要吗？我们甚至能想象出一个经常扫描数据库的自动化工具，寻找像这样的矛盾之处并修正它们。事实上，像这样的工具确实存在，并被用于一致性处于次要位置的数据库中。你甚至可能遇到过这样的例子，一些操作系统在崩溃后重启时，会检查整个文件系统的不一致性。

但确实存在不一致性非常有害且不能为自动化工具纠正的情况。在银行账户间转钱就是个经典例子。下面是另一个简单数据库：

| 账户名称 | 账户类型 | 账户余额 |
| --- | --- | --- |
| 扎迪 | 支票 | $ 800 |
| 扎迪 | 存款 | $ 300 |

| 皮德罗 | 支票 | $ 150 |
|---|---|---|

假设扎迪要求从她的支票账户转200美元到她的存款账户。就和前面的例子一样，这需要更新两行，会连续用到两次单独的磁盘操作。首先扎迪的支票余额会减少到600美元，其次她的存款余额会增加到500美元。而如果我们在这两次操作间遭遇崩溃，数据库看起来会这样：

| **账户名称** | **账户类型** | **账户余额** |
|---|---|---|
| 扎迪 | 支票 | $ 600 |
| 扎迪 | 存款 | $ 300 |
| 皮德罗 | 支票 | $ 150 |

换句话说，这对扎迪来说完全是一场灾难：在崩溃前，扎迪的两个账户一共有1100美元，但现在她只有900美元了。她从未取过钱，但200美元就这么完全消失了！而且还没有办法侦测到这一情况，因为数据库在崩溃后完全自相一致。在这里，我们遇到了一种更为细微的不一致性：新数据库与其崩溃前的状态不一致。

这点很重要，也值得更细致地进行研究。在第一个不一致性的例子里，我们最终得到的数据库自证为不一致：A和B成为朋友，但B却没有和A成为朋友。这种不一致性只需通过检查数据库就能侦测到（尽管侦测过程可能会非常耗时，如果数据库包含数百万乃或数十亿条记录的话）。在第二个不一致性的例子里，如果把数据库所处的状态当作某个时刻的截图，那么这一状态完全合理。没有规则规定这些账户的余额必须是多少，或规定余额之间存在任何关系。不过，如果我们按时间顺序检查数据库状态，就能观察到不一致行为。有三个事实和这一情况有关：（1）在开始转账前，扎迪有1 100美元；（2）在崩溃后，她有900美元；（3）在干扰期间，她并未取钱。这三个事实

放在一起就显得不一致，但不一致性不能通过在特定时刻检查数据库侦测到。

为避免这两种不一致性，数据库研究人员们提出了“事务”的概念——如果想让数据库最后保持一致性，就必须在数据库上完成一系列更改。如果一次事务中只执行了一些更改，那么数据库最后就可能不一致。这一想法很简单，但极其有效。数据库程序员可以发出如“begin transaction”（开始事务）这样的指令，然后对数据库做出大量互相依赖的更改，并以“end transaction”（终止事务）指令完成。数据库能保证程序员的更改都会完成，即便运行数据库的计算机在事务中途崩溃并重启。

为确保绝对正确，我们还应该意识到另一种可能性的存在：在崩溃并重启后，数据库也可能返回事务开始前的状态。但如果出现了这种情况，程序员会收到一条事务失败，必须重新提交的通知，以防造成任何损害。我们稍后将更细致地讨论这种可能性，在关于“回滚”事务（rolling back transaction）部分讲到。至于现在，关键在于不管事务是否完成还是回滚，数据库仍然保持一致性。

就目前的描述来看，我们似乎没有必要痴迷于崩溃的可能性。毕竟，崩溃在运行现代应用程序的现代操作系统上极少发生。对这一问题有两种回答。首先，这里所说的“崩溃”相当宽泛：包括任何可能导致计算机停止运作进而损失数据的事件。可能的事件包括断电、硬盘出错、其他硬件出错以及操作系统或应用程序中的漏洞。其次，即便这些泛指的崩溃极少发生，一些数据库也不能承受崩溃的风险：银行、保险公司和其他数据代表实际金钱的组织，这些组织不能承受任何情况下记录中出现不一致性的风险。

上面描述的解决方案的简洁性（开始一笔事务，执行必要的操作，然后终止事务）听起来也许太过美好，不能成真。事实上，下一部分描述的 “待办事项列表”把戏就相对简单，它也能完成处理事务。

## 待办事项列表把戏

并不是所有人都井井有条。但不管我们是否井井有条，我们都见过非常有条理的人使用的强力武器之一：待办事项列表。也许你并不喜欢制作列表，但很难质疑其有用性。如果你在一天中有10项任务要完成，把它们写下来——最好按完成效率排序——就是个非常好的开端。当你在工作日中间时刻分心（是否该说“崩溃”？）时，一张待办事项列表会尤其有用。如果你不知为何忘记了剩下的任务，扫一眼待办事项列表就能让你记起。

数据库事务要使用一种特殊的待办事项列表来完成。这也是我们称之为“待办事项列表”把戏的原因，尽管计算机科学家用“预写日志记录”这一术语称呼它。待办事项列表的基本思想是，维护一个数据库计划采取的动作日志。日志存储在硬盘或其他一些永久性存储介质中，这样，日志中的信息就能幸免于崩溃和重启。在一项事务的任何动作得到执行前，它们都被记录在日志中，然后再保存到磁盘里。如果事务成功完成，我们就能删除日志中的待办事项列表，进而节省一些空间。如此，上面描述的扎迪转账事务将分两步进行。首先，下图左侧的数据库表不动，我们在日志中写下本次事务的待办事项列表：

| **账户名称** | **账户类型** | **账户余额** |
|---|---|---|
| 扎迪 | 支票 | $ 800 |
| 扎迪 | 存款 | $ 300 |
| 皮德罗 | 支票 | $ 150 |

| **预写日志** |
|---|
| 1. 开始转账事务 |
| 2. 将扎迪的支票账户余额从800美元变为600美元 |

| 3.将扎迪的储蓄账户余额从300美元变为500美元 |
| --- |
| 4. 终止转账事务 |

在确保日志项保存到了某种永久性存储介质（如磁盘）上后，我们对表进行计划好的变更：

| 账户名称 | 账户类型 | 账户余额 |
| --- | --- | --- |
| 扎迪 | 支票 | $ 600 |
| 扎迪 | 存款 | $ 500 |
| 皮德罗 | 支票 | $ 150 |

| 预写日志 |
| --- |
| 1. 开始转账事务 |
| 2. 将扎迪的支票账户余额从800美元变为600美元 |
| 3.将扎迪的储蓄账户余额从300美元变为500美元 |
| 4. 终止转账事务 |

假设变更已保存到磁盘，日志项就能被删除了。

但这个例子很容易。假如计算机在事务中途意外地崩溃了呢？和前面一样，让我们假设崩溃发生在扎迪的支票账户被扣除款项之后，储蓄账户增加款项之前。计算机重启，数据库重启后在硬盘中发现如下信息：

| 账户名称 | 账户类型 | 账户余额 |
| --- | --- | --- |
| 扎迪 | 支票 | $ 600 |
| 扎迪 | 存款 | $ 300 |
| 皮德罗 | 支票 | $ 150 |

| 预写日志 |
| --- |
| 1. 开始转账事务 |
| 2. 将扎迪的支票账户余额从800美元变为600美元 |
| 3.将扎迪的储蓄账户余额从300美元变为500美元 |
| 4. 终止转账事务 |

## 大大小小的原子性

理解事务还有另外一种途径：从数据库用户的观点来看，每一笔事务都是原子态（atomic）。尽管物理学家们许多年前就知道了如何拆散原子，“atomic”的原意来自希腊，意味着“不可分割”。当计算机科学家说“原子态”时，他们指的是原意。因此，一笔原始态的事务不能被分成更小的操作：要么整笔事务成功地完成，要么数据库处于其原始状态，就像事务从未开始一般。

待办事项列表把戏给了我们原子态事务，反过来又保证了一致性。这是我们典型例子——一个针对在线银行的高效、完全可靠的数据库——中的一个关键要素。不过，我们还未达到那一步。一致性并不提供足够的高效性或可靠性。当和我们很快就会谈到的锁定技术（locking technique）结合时，待办事项列表把戏在数千名消费者同时连接数据库时都能保持一致性。这的确会提高效率，因为许多消费者能同时得到服务。而且待办事项列表把戏还提供了一种衡量可靠性的好方法，因为它能阻止不一致性。特别是，待办事项列表把戏排除了数据损坏，但并未消除数据丢失。下一个数据库把戏——预备提交把戏——将为朝向阻止任何数据丢失的目标做出巨大进步。

# 用于复制数据库的预备提交把戏

我们游览精巧数据库技术的旅程将继续，接下来我们将遇到名为“预备提交把戏”的算法。要了解这一把戏的动机，我们需要理解两个与数据库有关的事实：首先，它们会被复制，也就是说同一数据库的多份拷贝被存储在不同地点；其次，有时候数据库事务必须被取消，这被称为“回滚”或“放弃”一次事务。在转入预备提交把戏之前，我们将简短地了解这两个概念。

# 复制数据库

通过完成或回滚崩溃前正在进行的事务，待办事项列表把戏允许数据库从特定种类的崩溃中恢复。但前提是假设所有在崩溃前保存的数据都还在。假如计算机硬盘永久损坏，其中部分或所有数据都丢失了呢？这只是计算机永久丢失数据的途径之一。其他原因包括软件漏洞（数据库程序本身或操作系统漏洞）和硬件出错。这些问题都能导致计算机覆盖你认为安全地存储在硬盘上的数据，抹掉并用垃圾代替。很明显，待办事项列表把戏在出现这种情况时不能帮助我们。

然而，在一些情况中根本不能出现数据丢失。如果银行丢失了你的账户信息，你会极度郁闷，银行可能会面临严重的法律和财务惩罚。这也适用于证券交易公司，假如它在执行你下的订单时丢失了交易细节。的确，任何有可观网络销售额的公司（易贝和亚马逊就是最大的例子）都承担不起损失或损坏任何消费者信息的后果。但在有成千上万台计算机的数据中心里，每天都有许多组件（特别是硬盘）出错。这些组件上的数据每天都丢失。银行如何能在面临这种数据丢失的惨状下保持你的数据安全呢？

显然同时也广泛使用的解决方案是，保有两份及以上的数据库拷贝。每份数据库拷贝都被称为复制品（replica），所有拷贝的集合被称为复制数据库（replicated database）。通常，复制品在地理上分开（也许位于相隔数百英里的数据中心里），因此即便其中一份复制品被一场自然灾难抹掉，另一份复制品也还在。

我曾经听到一名计算机企业高管这么描述在2001年9月11日恐怖分子袭击纽约世贸中心双子塔后该公司客户的经历。那家计算机企业有五个主要客户在双子塔中，所有客户都运行着地理上分开的复制数据库。其中四位客户能依靠剩下的数据库拷贝基本不受干扰地继续开展业务。不幸的是，第五位客户在两座双子塔中都有一份复制品，但这两份都丢失了！这位客户只能在从站外档案备份中恢复数据库后，才得以重新开展业务。

注意，复制数据库和“备份”一些数据的常见概念有很大不同。备份是某个特定时刻对一些数据的快照——对于手动备份，快照在你运行备份程序时进行，而自动备份通常会在每周或每天的某个特定时刻对系统进行快照，比如每天早晨2点钟。换言之，一个备份是一些文件、一个数据库或其他任何你需要额外拷贝的东西的完全拷贝。

不过，根据定义，备份并不一定最新：如果在一次备份之后做出一些变更，这些变更不会被保存在备份中。相反，复制数据库随时保持数据库的所有拷贝同步。每次数据库中任一项哪怕出现最细微的变更，所有复制品都必须立即做出变更。

很明显，复制是抵御数据丢失的绝佳方法。但复制也存在危险：它引入了另一种可能的不一致。如果一个复制品最后的数据和另一个复制品不同，这时我们该怎么办？这样的复制品彼此不一致，很难或不可能判定哪个复制品中是正确的数据。在研究了如何回滚事务后，我们将再返回讨论这一问题。

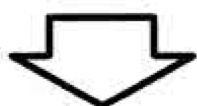
## 回滚事务

冒着重复的风险，让我们尝试回忆一下事务究竟是什么：事务是对数据库的一系列变更，这些变更必须全部执行以保证数据库保持一致。在之前对事务的讨论中，我们主要关注事务会完成，即便数据库在事务进行途中崩溃。

但结果证明，有时候出于一些原因不可能完成一项事务。比如，也许事务涉及向数据库中添加大量数据，而计算机在事务中途用完了磁盘空间。这一情况非常罕见，但却很重要。

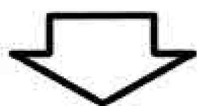
另一个更常见的未能完成事务的原因和另一个名为锁定（locking）的数据库概念有关。在一个繁忙的数据库中，通常会同时执行多项事务。（想象一下，假如银行每次只允许一名用户转账，这时会发生什么情况——这种在线银行系统的效能会让人无比震惊。）在一次事务进行中，数据库的一些部分会冻结，这点很重要。比如，如果事务A向记录中上传一个罗西娜和静宜成为朋友的项，而同时事务B从数据库中删除了静宜，这会造成灾难性的后果。因此，事务A将“锁定”包含静宜信息的那部分数据库。这意味着那些数据被冻结了，其他事务不能改变它。在大多数数据库中，事务能锁定单行或单列，或整个表。

| 名字 | 年龄 | 电子邮件 |
| --- | --- | --- |
| ⋮ | ⋮ | ⋮ |
| 玛丽 | 21 | mb@abc.edu |
| ⋮ | ⋮ | ⋮ |
| 皮德罗 | 47 | pedro@xyz.org |



| 名字 | 年龄 | 电子邮件 |
| --- | --- | --- |
| ⋮ | ⋮ | ⋮ |
| 玛丽 | 21 | mb@abc.edu |
| ⋮ | ⋮ | ⋮ |
| 皮德罗 | 47 | pedro@xyz.org |

A
B



| 名字 | 年龄 | 电子邮件 |
| --- | --- | --- |
| ⋮ | ⋮ | ⋮ |
| 玛丽 | 21 | mb@abc.edu |
| ⋮ | ⋮ | ⋮ |
| 皮德罗 | 47 | pedro@xyz.org |

A ← ?! B
B ← ?! A

死锁（deadlock）：当两次事务A和B同时尝试锁定同一行——但方向相反——时，它们就会死锁，都不能进行。

很明显，一次只能有一项事务锁定数据库的某个部分。一旦该项事务成功完成，就会“解锁”之前被它锁定的所有数据。之后，其他事务就能更改之前被冻结的数据。

乍一看，这似乎是一个绝佳的解决方案，但这会导致一个非常糟糕的状况，计算机科学家称之为死锁，正如上图所示。让我们假设两项事务A和B同时运行。正如上图顶部所示，最初数据库中没有行被锁住。之后，正如上图中部所示，A锁住了包含玛丽信息的那一行，B锁住了包含皮德罗信息的那一行。过后不久，A发现需要锁住皮德罗所在的行，B发现需要锁住玛丽所在的行——这一情形如上图底部所示。注意，A现在需要锁住皮德罗所在的行，但一次只能有一项事务锁住任意行，但B已经锁住了皮德罗所在的行！因此A需要等到B结束才能锁定。但B只有在锁定玛丽所在的行时才能结束，而玛丽所在的行目前又由A锁定。因此B需要等到A结束才能锁定。A和B互为死锁，因为每项事务都必须等待另一项事务才能进行。它们会永远僵持下去，这些事务永远不会完成。

计算机科学家已经非常细致地研究了死锁，许多数据库都会定期运行侦测死锁的特殊程序。当发现一个死锁时，其中一项死锁的事务会取消，以便让另一项事务进行。但请注意，就和在事务进行途中耗尽磁盘空间时所做的一样，这要求具备撤销或“回滚”已经部分完成的事务的能力。现在我们至少知道了两种事务需要回滚的原因。还有许多其他原因，但我们在此无须深入。根本点在于，事务经常因为不可预料的原因而不能完成。

回滚能通过对待办事项列表把戏稍作变更来实现：预写日志必须包含足够的额外信息，才能在必要时撤销每次操作。（这与之前的描述相矛盾，我们在前面强调每一个日志项要包含足够多信息，以便在

崩溃后重新执行操作。）这在实践中很容易达成。事实上，在我们审视的简单例子中，撤销信息和重新执行的信息都一样。像“将扎迪的支票账户余额从800美元变为600美元”这样的项能轻易“撤销”——只需将扎迪的支票账户余额从600美元变为800美元。总之：如果一项事务需要回滚，数据库程序只需通过预写日志（比如待办事项列表）逆向操作，逆转事务中的每项操作。

## 预备提交把戏

现在，让我们思考复制数据库中回滚事务带来的问题。这里的问题是：其中一份复制品可能会遇到了要求回滚的问题，而其他复制品则没有这样的问题。比如，假设其中一份复制品耗尽了磁盘空间，而其他复制品仍有空间剩余。

一个简单的类比在这里会很有帮助。假设你和三个朋友想一起去看一部最近上映的电影。为了让例子更有趣，让我们把故事背景设在20世纪80年代，那时还没有电子邮件，因此电影之行将通过电话组织。你会怎么组织？下面列出了一种可能的方法：确定一个放映电影的时间，这个时间要适合你，就你所知这个时间极有可能也适合你朋友。假设你选择了周二下午8点。下一步就是打电话给其中一位朋友，询问他或她在周二下午8点是否有空。如果有空，你会说“太好了，用笔记下来，我等下再打回来确认”。然后你再打电话给下一位朋友，做相同的事。最后，你打电话给第三位以及最后一位朋友，提出相同的问题。如果所有人都在周二晚上8点有空，你就能做出最终决定，确认这一事件，并打电话给朋友，让他们知晓。

这是简单的情形。假如一位朋友在周二晚上8点没空呢？那样的话，你需要“回滚”已经做了的所有工作，重新开始。在现实中，你可能会打电话给每个朋友，并立即给出一个新时间，但为了让事情尽可

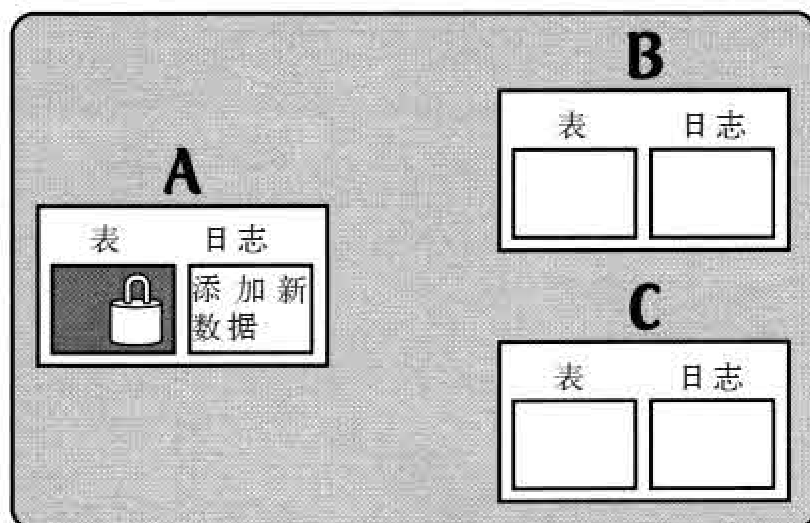
能简单，假设你打电话给每位朋友并说“不好意思，周二晚上8点不合适，在你的日程上删掉它把，我会很快打电话告诉你新时间”。一旦做完这件事情，你就能重新开始整个流程。

注意，在你安排电影出行的策略中有两个截然不同的阶段。在第一阶段，你提出了日期和时间，但并非百分之百地确定。一旦你发现提议适用所有人，你才能百分之百地确定日期和时间，但其他人不知道。因此，你在第二阶段给所有朋友打电话回去确认。相反，如果一个或更多朋友不能成行，第二阶段就包括给所有人打电话取消。计算机科学家们将之称为两阶段提交协议；我们称之为“预备提交把戏”。第一阶段被称为“预备”阶段。第二阶段要么是“提交”阶段，要么是“撤销”阶段，取决于最初提议是否被所有人接受。

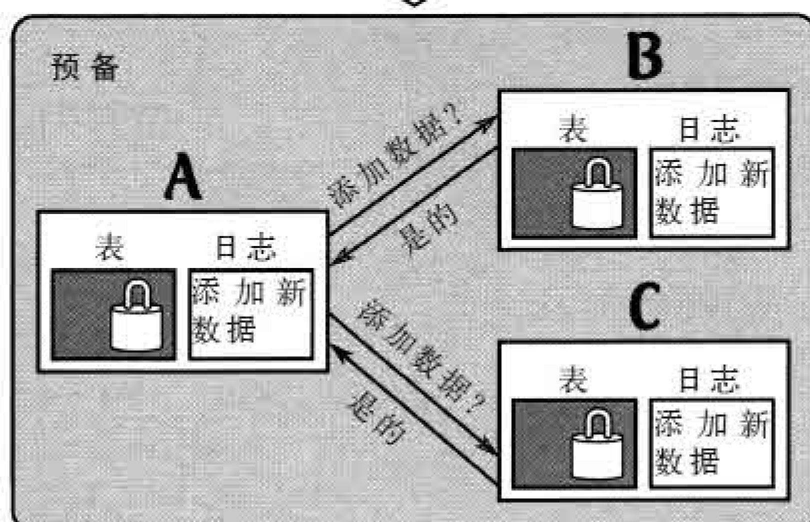
有趣的是，这一类比中有个数据库锁定的想法。尽管我们并未详尽讨论它，你所有的朋友在写下电影出行时都做了一个不详尽的承诺：他们承诺不会在周二晚8点安排其他事。直到他们收到确认或取消的回信为止，日程表上的那个时间段都是“锁定”的，不能被其他事务“更改”。比如，如果其他人打电话给你朋友，时间恰好是第一通电话和第二通电话之间，提议在周二晚8点观看一场篮球赛呢？你朋友应该会说，“对不起，但我可能在那个时间有另一场约会。在那场约会敲定前，我不能给你确信是否去看这场篮球赛。”

现在，让我们审视一下预备提交把戏如何在复制数据库中起作用。下图展示了这一想法。一般来说，其中一个复制品是协同事务的“主管”（master）。更确切点讲，假设有三个复制品A、B和C，A是主管。假设数据库需要执行一项向表中插入一行新数据的事务。预备阶段从A锁定表开始，接下来A将新数据写入其预写日志。同时，A将新数据发给B和C。B和C也都会锁定各自的表拷贝，并在日志中写入新数据。然后B和C向A返回报告它们是否成功地做了这一事务。接下来第二阶段开始，如果A、B或C中任一项事务遇到了一个问题（比如

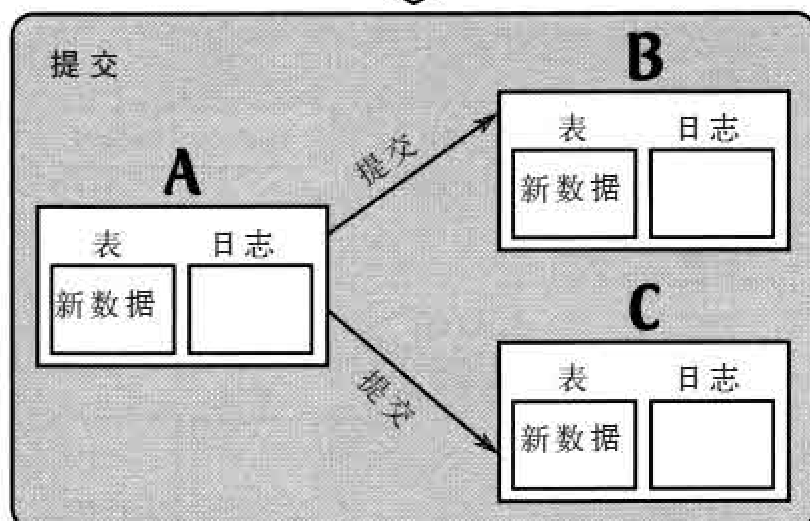
耗尽磁盘空间或未能锁定表），主管A就知道事务必须回滚，并通知所有复制品这一点。但如果所有复制品都在预备阶段报告成功，A会向每份复制品发送消息确认事务，复制品接下来会完成事务（如下图所示）。


B
表
日志
A
表
日志
添加新数据
C
表
日志

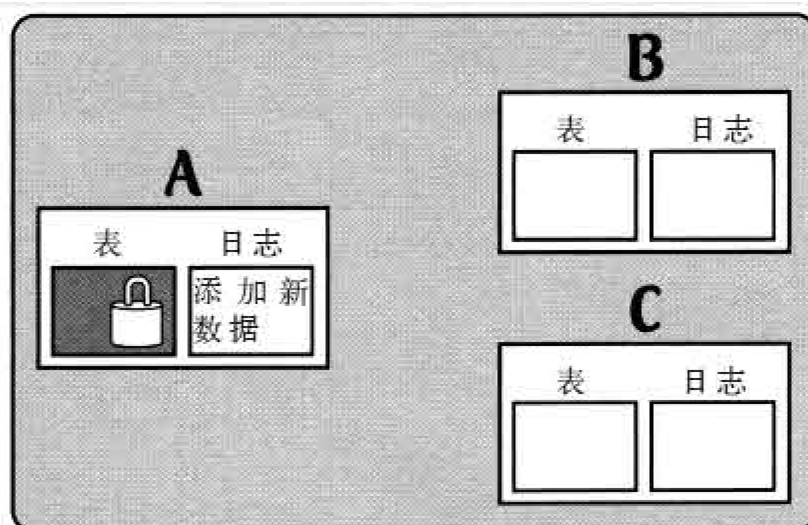



预备
B
A
添加数据?
是的
表
日志
添加新数据
C
添加数据?
是的



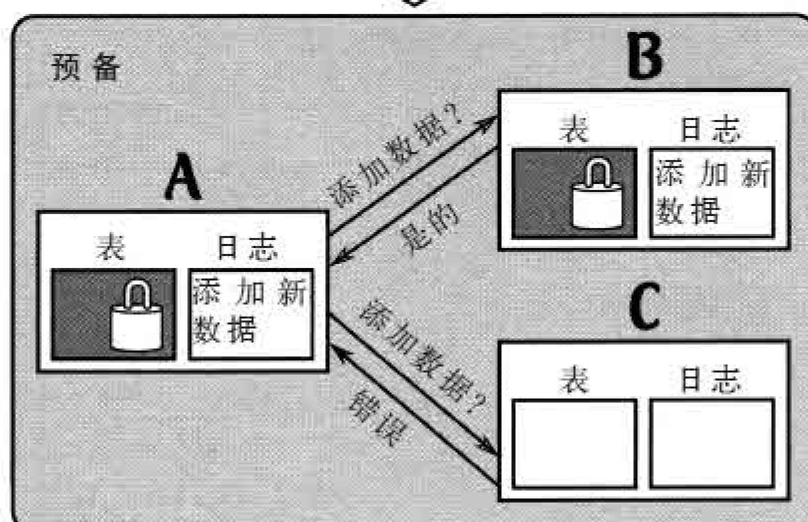

提交
B
A
提交
表
日志
新数据
C
提交

预备提交把戏：主管复制品A协同另外两个复制品（B和C）向表中添加一些新数据。在预备阶段，主管复制品检查是否所有复制品都能完成事务。一旦所有事情都妥当，主管复制品会让所有复制品提交数据。



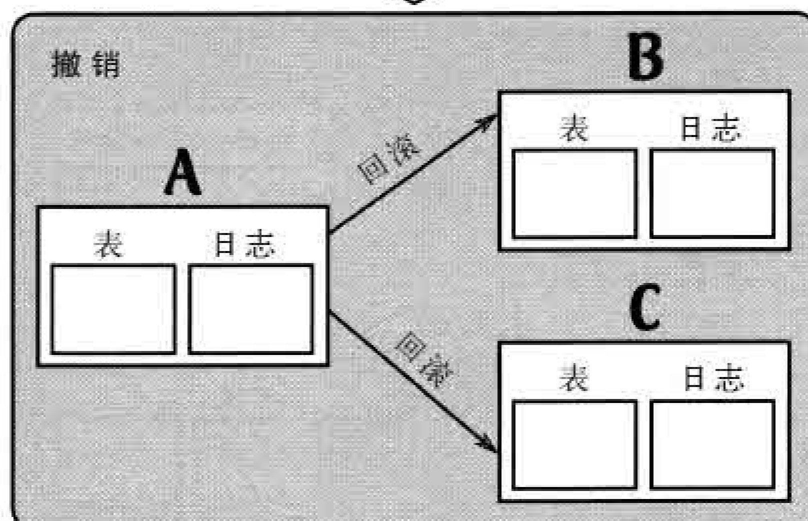
A
表
日志
添加新数据
B
表
日志
C
表
日志



预备
A
表
日志
添加新数据
添加数据？
是的
添加数据？
错误
B
表
日志
添加新数据
C
表
日志



撤销
A
表
日志
回滚
回滚
B
表
日志
C
表
日志

带回滚操作的预备提交把戏：本图顶部和上一张图顶部一样。但在预备阶段，其中一个复制品出错。结果，本图底部是“撤销”阶段，其中每个复制品都必须回滚事务。

到目前为止，我们的处理方式中有两种数据库把戏：待办事项列表把戏和预备提交把戏。它们对我们有什么用？通过综合这两个把戏，银行——以及其他任何在线实体——都能使用原子态事务布置复制数据库。而这能在同时为成千上万的客户提供高效服务，且基本上不会出现不一致或数据丢失。不过，我们还未深入数据库的核心：数据如何构建，查询如何得到回答？最后一个数据库把戏会为这些问题提供一些答案。

## 关系数据库和虚表把戏

在前面的所有例子中，数据库都只由一张表组成。但现代数据表技术的真正威力在有多张表的数据库中才释放出来。关系数据库的基本思想是，每张表都存储不同的信息集，但不同表中的个体通常都以某种方式相连。因此，一个公司的数据库也许是由不同的表组成，这些表包括客户信息、供应商信息和产品信息。但客户表可能会提及产品表的东西，因为客户会订购产品。也许产品表会提及供应商表中的东西，因为产品由供应商的商品制造。

让我们来看一个真实的小例子：由大学存储的信息详细列出了学生要上哪些课程。为便于管理，例子中只会有几位学生和几门课程，但希望你能明白，同一原理也适用于大得多的数据量。

首先，让我们看看数据会如何使用简单的一张表方式进行存储，我们在本章前面部分一直使用这一方法。整个过程在下面的顶图中显示。你可以看到，数据库中有10行和5列；衡量这个数据库信息量的一个简单方法是说，数据库中有10×5 = 50个数据项。花几秒钟更细致地研究下面的表。有什么东西让你对信息存储的方式感到恶心吗？比

如，你能看到任何不必要的数据重复吗？你能想到存储相同信息更有效的方式吗？

你可能已经意识到，有关每门课程的大量信息都复制给了参加该门课程的所有学生。比如，三名学生参加ARCH101课程，而关于这门课程的细节信息（包括课程名、讲师和房间号）都向每位学生重复了。存储这些信息更高效的方法是使用两张表：一张表存储学生们上哪些课，另一张表存储与每门课程有关的细节。这种双表策略在下面的底图中显示。

| 学生名 | 课程号 | 课程名 | 讲师 | 房间号 |
|---|---|---|---|---|
| 弗朗西斯卡 | ARCH101 | 考古学入门 | 布莱克教授 | 610 |
| 弗朗西斯卡 | HIST256 | 欧洲历史 | 史密斯教授 | 851 |
| 苏珊 | MATH314 | 微分方程 | 科比教授 | 560 |
| 埃里克 | MATH314 | 微分方程 | 科比教授 | 560 |
| 路易吉 | HIST256 | 欧洲历史 | 史密斯教授 | 851 |
| 路易吉 | MATH314 | 微分方程 | 科比教授 | 560 |
| 比尔 | ARCH101 | 考古学入门 | 布莱克教授 | 610 |
| 比尔 | HIST256 | 欧洲历史 | 史密斯教授 | 851 |
| 罗斯 | MATH314 | 微分方程 | 科比教授 | 560 |
| 罗斯 | ARCH101 | 考古学入门 | 布莱克教授 | 610 |

| 学生名 | 课程号 |
|---|---|
| 弗朗西斯卡 | ARCH101 |
| 弗朗西斯卡 | HIST256 |
| 苏珊 | MATH314 |
| 埃里克 | MATH314 |
| 路易吉 | HIST256 |
| 路易吉 | MATH314 |

| | |
|---|---|
| 比尔 | ARCH101 |
| 比尔 | HIST256 |
| 罗斯 | MATH314 |
| 罗斯 | ARCH101 |

| 课程号 | 课程名 | 讲师 | 房间号 |
|---|---|---|---|
| ARCH101 | 考古学入门 | 布莱克教授 | 610 |
| HIST256 | 欧洲历史 | 史密斯教授 | 851 |
| MATH314 | 微分方程 | 科比教授 | 560 |

顶图：学生课程的单表数据库
底图：用两张表更高效地存储了相同数据

我们马上能看到这种多表方法的好处之一：要求的存储总量减少了。新方法使用了一个10行2列的表（10×2 = 20个项）和一个3行4列的表（3×4 = 12个项），总共是32个项。相反，一表方法需要50个项才能存储相同多的信息。

节省是如何实现的呢？它来自于对重复信息的消除：和重复每位学生选取的每门课程的名称、讲师和房间号不同，这些信息只向每门课程列出了一次。尽管我们牺牲了一些东西来实现这一点：现在课程号出现在两个地方，两张表中都有一个“课程号”列。我们用大量重复（课程细节）和少量重复（课程号）进行了交换。总体而言，这是笔好交易。这个小例子中的收获并不大，但你也许想象得出，如果有成百上千的学生参加同一门课，通过这一方法将节省巨大的存储空间。

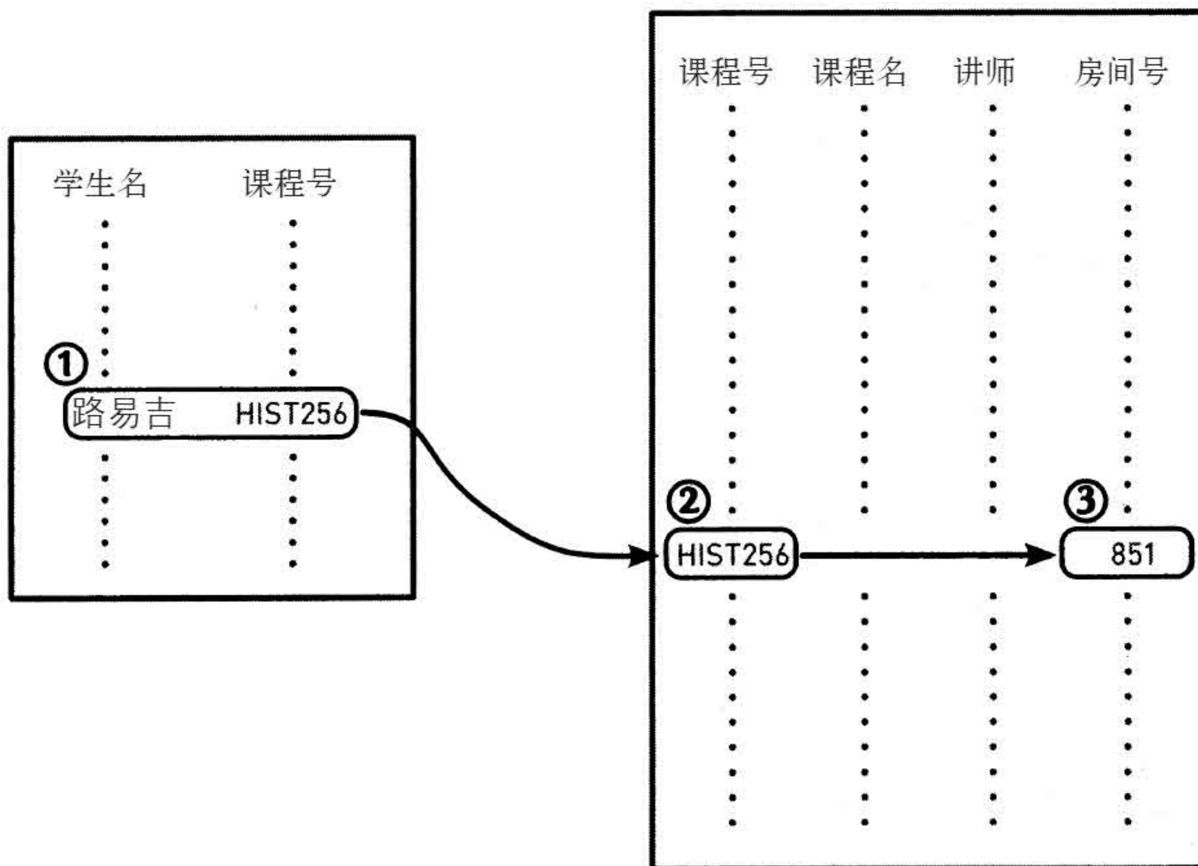
多表方法还有另一个巨大优势。如果表设计无误，对数据库的变更会更容易。比如，假设课程MATH314的房间号从560变为440。在一表方法中，有四个单独的行需要更新。而且，正如我们之前讨论的，

这四次更新需要包含在一次事务中，以确保数据库保持一致。但在多表方法中，只需要进行一次更改——更新课程细节表中一项即可。

## 键

在这里有一点值得指出，这个简单的学生课程例子使用两个表来代表最高效，真实的数据库通常会和许多表协同。很容易想象用新表扩展学生课程的例子。比如，可能有一张包含每位学生细节的表，如学生号、电话号以及家庭住址。可能还有一张讲师表，列出讲师的电子邮箱地址、办公地点和办公时间。每个表都被设计用其大多数列存储数据，以便让这些数据不在其他地方出现——这种方法的思想是，不管何时要求某个特定物体的细节，我们都能在相关表中“查询”这些细节。

在数据库术语中，表中用于“查询”细节的列被称为“键”。比如，想象一下如何寻找路易吉的历史课房间号。通过使用上一节所述的单表方法，我们只需扫描表中的行，直到找到路易吉的历史课，然后查询房间号列，得到结果851。但在多表方法中，我们最开始扫描第一个表，以发现路易吉的历史课课程号——也就是“HIST256”。然后我们将“HIST256”作为其他表的键使用：通过寻找包含课程号“HIST256”的行，查询这门课程的细节，然后在该行中寻找房间号（即851）。这一过程在下图中显示。




用键查找数据：为找到路易吉历史课的房间号，我们首先要找到左表中的相关课程号。然后这个值“HIST256”会被用作另一张表的键。因为课程号栏以字母表顺序排序，我们能很快找到正确的行，然后获取对应的房间号（851）。

像这样使用键的美妙之处在于，数据库能以超高效率查询键。这和人在字典中查询一个单词类似。想想你会如何在纸质词典中查找“epistemology”（认识论）这个单词。自然地，你不会从第一页开始，扫描每个项以查找“epistemology”。相反，你很快就能通过查看页首来缩小范围，你最开始会大块翻页，然后会慢慢放慢翻页的幅度，逐渐靠近你的目标。数据库使用同样的技术查找键，但它们要比人高效得多。这是因为数据库能提前计算出需要翻多少“块”页，并能记录每块开始和结束的页首。计算机科学中有一种用于快键查找的预计算块集合被称为B树（B-tree）。B树是另一种支持现代数据库的关键精巧思想，但不幸的是，对B树进行细致讨论会离题太远。

## 虚表把戏

我们准备好去领会多表数据库背后精巧的主要思想了。虚表的基础思想很简单：尽管所有的数据库信息都能存储在一个固定大小的表中，数据库能在需要时生成新的临时表。我们称这些表为“虚表”，以强调它们不会被存储到任何地方的事实——当数据库在回答对数据库的查询并且需要虚表时就会创建它们，并立即删除它们。

举个简单例子来说明虚表把戏。假设我们从前文底图描述的数据库开始，一名用户输入一个查询，询问所有上科比教授课的学生名字。数据库处理这一查询的方法有许多种；我们会审视其中一种方法。第一步是创建一张新虚表，列出所有课程的学生和讲师。这可以通过合并（join）——一种特殊的数据库操作——两个数据库来完成。合并的基本思想是将一个表中的每一行和另一张表对应行结合起来，对应是通过同时出现在两张表中的键栏建立起来的。比如，当我们合并第前文底图两张表时，会使用“课程号”作为键，结果就是一张如前文顶图一样的虚表——每个学生都和第二张表中相关课程的所有细节相结合，人们可以使用“课程号”作为键查找这些细节。当然，原始查询和学生们及讲师有关，因此我们不需要其他栏。幸运的是，数据库包含一个抛射（projection）操作，能让我们抛弃不感兴趣的栏。因此在将两张表结合的合并操作后，会有抛射操作移除一些不必要的栏，最终数据库得到了下面的虚表：

| 学生名 | 讲师 |
| --- | --- |
| 弗朗西斯卡 | 布莱克教授 |
| 弗朗西斯卡 | 史密斯教授 |
| 苏珊 | 科比教授 |
| 埃里克 | 科比教授 |
| 路易吉 | 史密斯教授 |

| 路易吉 | 科比教授 |
| --- | --- |
| 比尔 | 布莱克教授 |
| 比尔 | 史密斯教授 |
| 罗斯 | 科比教授 |
| 罗斯 | 布莱克教授 |

接下来，数据库会使用另一种名为选取（select）的重要操作。选取操作会基于一些标准，从一张表中选取一些行，并抛弃其余的行，生成一张新虚表。在这个例子中，我们在寻找上科比教授课的学生，因此我们要进行一次“选取”操作，只选择讲师为“科比教授”的行，得到的虚表如下：

| **学生名** | **讲师** |
| --- | --- |
| 苏珊 | 科比教授 |
| 埃里克 | 科比教授 |
| 路易吉 | 科比教授 |
| 罗斯 | 科比教授 |

查询快完成了。现在我们需要的是另一次抛射操作，抛弃“讲师”栏，余下一张回答原始查询的虚表：

| **学生名** |
| --- |
| 苏珊 |
| 埃里克 |
| 路易吉 |
| 罗斯 |

我认为值得在此添加一个略显技术性的提示。如果你恰巧对数据库查询语言SQL很熟悉，你也许会觉得上面对“选取”操作的定义相当

奇怪，因为SQL中的“选取”命令功能比只选取一些行大得多。这一术语来自于一种数据库操作数学理论，也就是知名的关系代数（relational algebra）,其中的“选取”仅用于选取行。关系代数还包含“合并”和“抛射”操作，这两项操作在寻找科比教授学生的查询中用到过。

## 关系数据库

一个将其所有数据都存储在我们之前用过的互联表中的数据库被称为关系数据库。关系数据库由IBM研究员埃德加·科德（E. F. Codd）在其于1970年发表的极具影响力的论文《大型共享数据库数据的关系模型》（A Relational Model of Data for Large Shared Data Banks）中提出。和科学上许多伟大思想一样，回过头看关系数据库似乎很简单——但在那时，它们代表了在高效存储和处理信息上的巨大飞跃。结果显示，只需几种操作（如我们之前见过的关系代数操作“选取”、“合并”以及“抛射”）就足够生成虚表，回应几乎所有对关系数据库的查询。因此，关系数据库能将数据存储在为高效性而构建的表中，使用虚表把戏回应似乎需要在不同表中数据的查询。

这也是关系数据库被用于支持大多数电子商务活动的原因。当你网上购物时，你很有可能就和一系列关系数据库表进行了互动，这些表存储了与产品、客户和单次购买相关的信息。在网络空间中，我们无时无刻不被关系数据库所包围，很多时候甚至没有意识到它们的存在。

## 数据库的人性面

对于一般的旁观者而言，数据库也许是本书中最不令人激动的主题。要为数据存储感到激动很难。但在背后，让数据库奏效的精巧思

想却是另一回事。建立在能在任何操作中途出错的硬件上，数据库给予我们期望在线银行及类似活动能拥有的高效性和坚实可靠性。待办事项列表把戏给了我们原子态事务，即便成千上万的客户同时和一个数据库互动也能保持一致性。这种并发性的深厚程度，与虚表把戏提供的快速查询响应能力一道，让大型数据库高效化。待办事项列表把戏还保证了面临出错时的一致性。当与用于复制数据库的预备提交把戏结合时，我们就获得了牢靠的一致性以及数据的持久性。

数据库对不可靠组件的英勇胜利——计算机科学家称之为“容错”（fault-tolerance）——是许多研究人员在过去几十年间的成果。但其中最重要的贡献者是吉姆·格雷（Jim Gray）。格雷是位超级计算机科学家，撰写了有关事务处理的数据。[这本书名为《事务处理：概念与技术》（Transaction Processing: Concepts and Techniques），该书于1992年首次出版。]令人遗憾的是，他的职业生涯很早就结束了：2007年的一天，他驾驶自己的游艇由金门大桥出旧金山湾，进行一次计划好的白昼旅行，准备前往旧金山湾附近公海中的一些岛屿。自此以后，格雷和他的船音信全无。这个悲剧故事中有一处温暖人心的转折，格雷在数据库社区的许多朋友使用格雷的工具尝试拯救格雷：旧金山附近海域最新得到的卫星照片被上传到一个数据库中，以便让朋友们和同事们搜寻这位失踪数据库先锋的线索。不幸的是，这次搜寻并不成功，计算机科学世界丧失了一位领军人物。

# 第九章　数字签名——这个软件究竟由谁编写

为了证明你错得多厉害，你的假设是多么没有根据，我把证书摆在你面前……看看它！你可以拿在手里看；它绝非伪造。

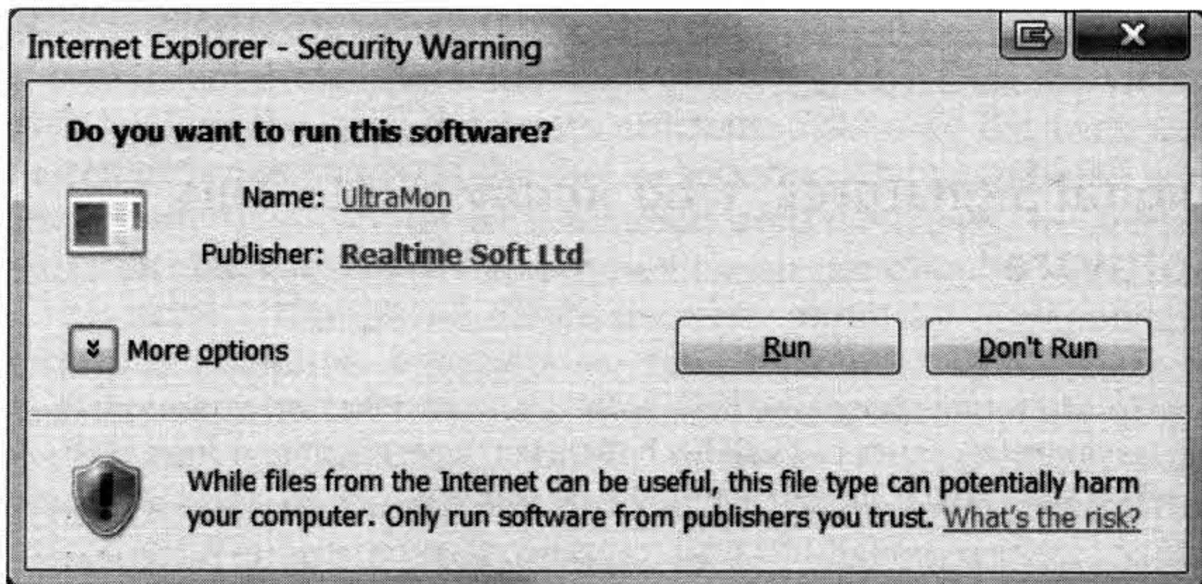
——查尔斯·狄更斯，《双城记》

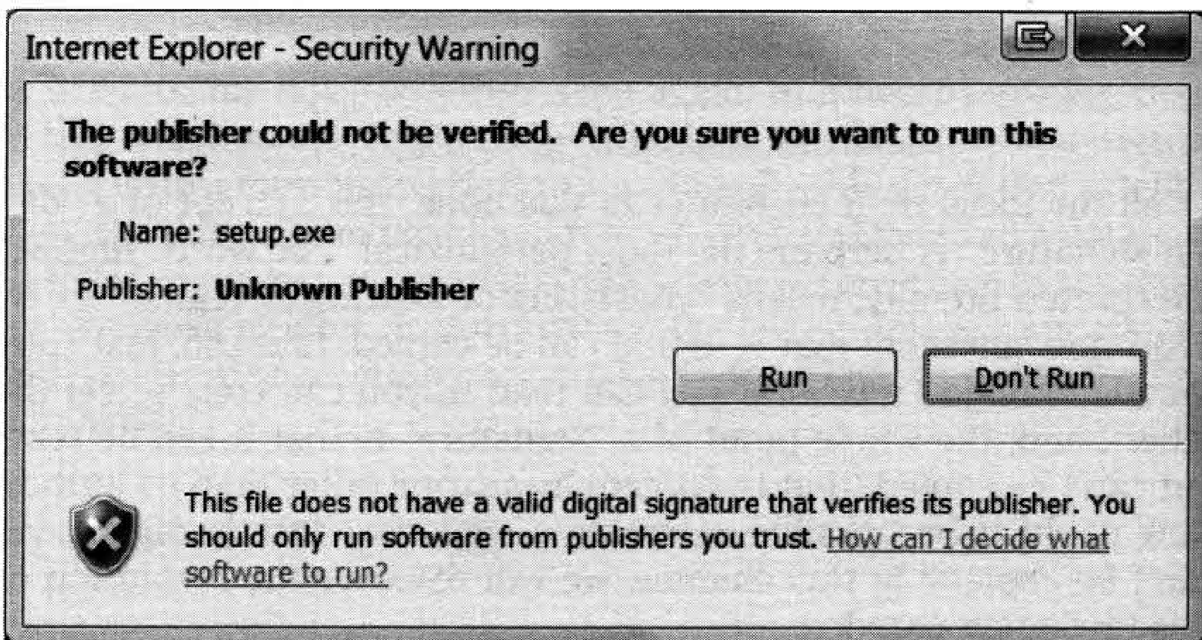
在本书的所有思想中，最矛盾的也许要数“数字签名”（digital signature）概念了。从字面来看，单词数字化（digital）数字化意味着其“由数字字符串组成”。因此，根据定义，任何数字化的东西都能被拷贝：要做到这一点，只要每次拷贝一下数字就行。如果你能读取数字，就能拷贝数字！另一方面，“签名”的全部意义在于能被读取，但不能被除作者外的任何人拷贝（也就是伪造）。怎么才有可能创造一个数字化的，但又不能被拷贝的签名呢？我们将在本章为这一有趣的悖论发现解决方案。

## 数字签名真正的用途有哪些？

数字签名的用途有哪些？问这个问题也许显得多余。你也许会想，我们能在纸质签名用到的地方使用数字签名：签署支票或其他法律文件，如出租一间公寓。但如果你思考一阵子，你会意识到这不对。当你进行在线支付时，不管是用信用卡还是通过在线银行系统，你会提

供任何种类的签名吗？没有。基本上，在线信用卡支付并不要求签名。在线银行系统略有不同，因为它们会要求你用密码登录，以帮助验证你的身份。但如果你稍后于在线银行对话期间进行支付，也不需要你提供任何种类的签名。




计算机会自动检查数字签名。
顶图：当我试图下载并运行一个拥有有效数字签名的程序时，网络浏览器显示的消息。
底图：当数字签名失效或缺失时，网络浏览器显示的消息。

那么，数字签名在实际生活中有哪些用途呢？答案可能和你一开始的想法相反：与你签署发送给其他人的材料不同，基本上是其他人先签署材料再发送给你。你很有可能并未意识到这一点的原因是，数字签名由计算机自动验证。比如，不管你何时想下载并运行程序，网络浏览器都会检查程序是否有数字签名以及数字签名是否有效。然后浏览器再显示一个合适的警告，如上图所示。

如你所见，这里有两种可能性。如果软件拥有有效签名（如上页顶图），计算机能完全肯定地告诉你编写该软件的公司名。当然，这并不能保证软件安全，但至少你能基于对该公司的信任程度做出充分了解后再做决定。反之，如果签名失效或缺失（如上页底图），你绝不会得到软件来自哪里的保证。即便你认为自己下载的软件来自一个信誉良好的公司，也有可能黑客用一些恶意软件替换了真正的软件。软件也有可能由业余人士编写，他们没有时间或动力去创造一个有效的数字签名。在这些情况下，是否安装取决于用户是否信任软件。

尽管软件签名是数字签名最明显的应用，但这绝不是仅有的一种应用。事实上，计算机接收和验证数字签名的频率非常多，因为一些经常使用的互联网协议也使用数字签名，验证与你进行交互的计算机的身份。比如，网络地址以“https”开头的安全服务器在建立一个安全对话前，通常会向你的计算机发送一个数字签名证书。数字签名还应用于验证许多软件组件的真实性，比如浏览器插件。你在浏览网页时很有可能已经见过这样的警告消息。

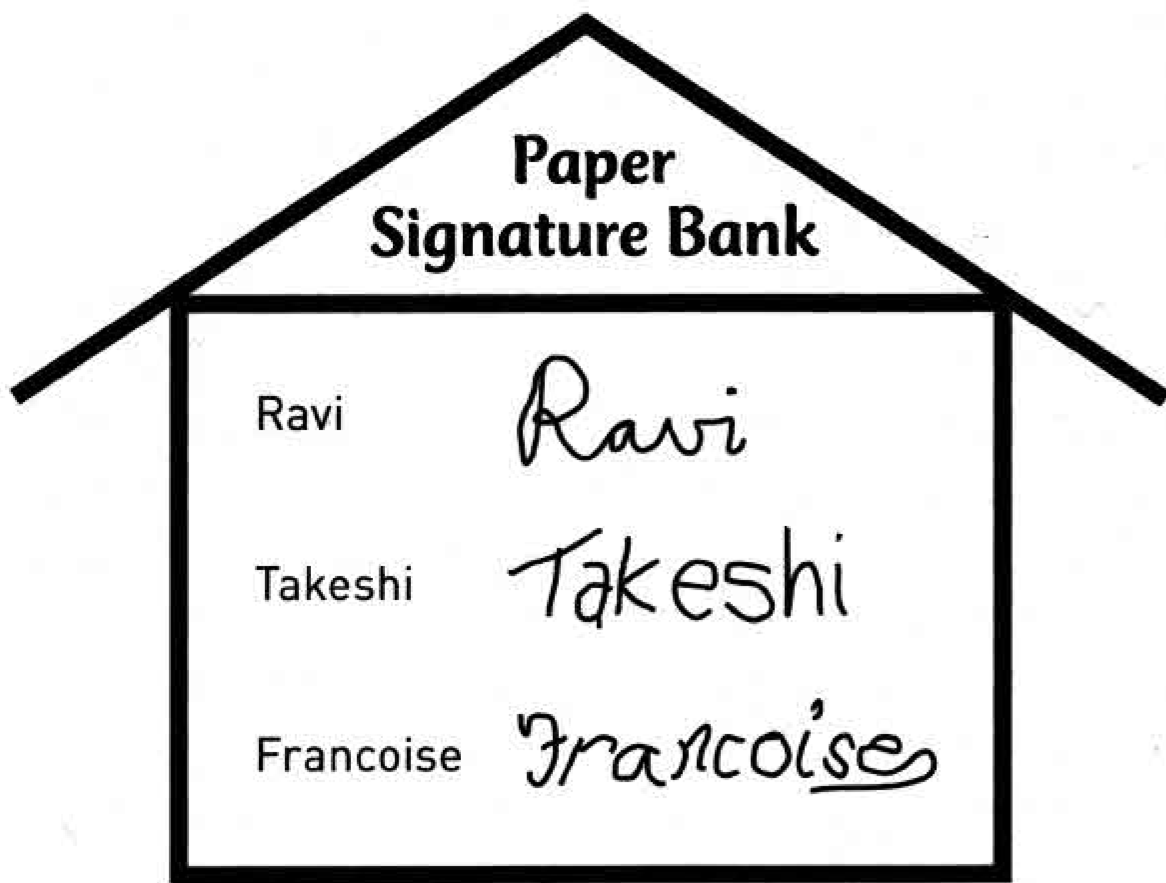
你也可能遇到另外一种网络签名：一些网站会要求输入名字，作为网络表格的签名。比如，当我发送为学生写的推荐信时，有时我必须这么做。这并不是计算机科学家所指的数字签名！很显然，伪造这种输入的签名毫不费力，任何知道你名字的人都能伪造。在本章，我们将学习如何创造一个不能被伪造的数字签名。

## 纸质签名

我们对数字签名的解释会一步步实现，先从纸质签名的熟悉场景开始，再慢慢转向精巧的数字签名。下面正式开始，让我们回到一个没有计算机的世界。在那个世界里，唯一确认文件可靠的方式就是纸质手写签名。注意，在这种场景下，一份已经签署的文件不能单独验证。比如，假设你找到的一张纸上写着“我承诺向弗朗索瓦丝支付100美元。签名，Ravi”——如下图所示。你如何才能验证拉维（Ravi）真的签署了这份文件呢？答案是你需要一些受信签名库，你可以在这个签名库中核实拉维签名的真实性。在现实世界中，银行、政府等机构会扮演这一角色——它们的确会保留存储有客户签名的文件，这些文件在必要时可进行物理核实。在我们的假想场景中，让我们假设有一个名为“纸质签名银行”（paper signature bank）的受信机构保留每个人的签名文件。下图就是一个纸质签名银行的概要图例。




一个用文件存储其客户身份及手写签名的银行。



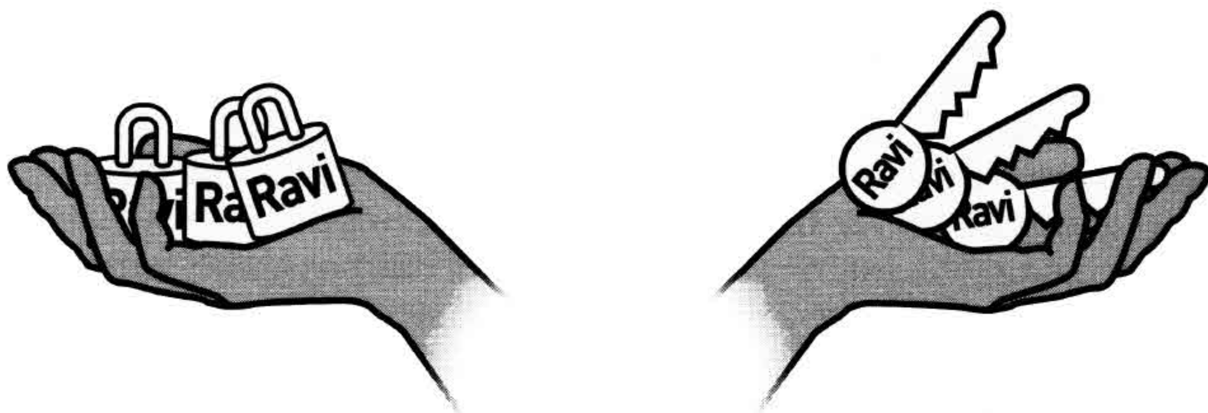

一份带有一个手写签名的纸质文件。

要验证拉维在承诺支付弗朗索瓦丝的文件上的签名，我们只需前往纸质签名银行，要求查看拉维的签名即可。很显然，我们在这里做出了两个重要假设。首先，我们假设能信任这家银行。理论上，该银行的员工可能将拉维的签名换成假冒者的签名，但我们在这里要忽略这种可能性。其次，我们假设假冒者不可能伪造拉维的签名。每个人都知道，这一假设错得离谱：熟练的造假者能轻易伪造签名，即便是业余人士，也能制作合理的相似签名。不管怎样，我们需要签名不可伪造这一假设——没有它，纸质签名就会毫无用处。我们将在后面知道，为何数字签名基本上不可能伪造。这也是数字签名对纸质签名的一大优势。

## 用挂锁签名

我们迈向数字签名的第一步是完全抛弃纸质签名，采用一种依赖挂锁、钥匙和已锁箱子的新方法验证文件。新机制的每位参与者（在这个例子中就是拉维<Ravi>、勇<Takeshi>和弗朗索瓦丝<Francoise>）都有大量挂锁供应。每位参与者拥有的挂锁都一样，因此拉维的挂锁都一样，这点很关键。另外，每位参与者的挂锁都必须是独有的：其他人不能制作或获得拉维的挂锁。最后，本章所有挂锁都具有一个相当不同寻常的特性：它们装备有生物传感器，以确保只有它们的所有者能上锁。如果弗朗索瓦丝发现有一把拉维的挂锁没锁上，她不能用拉维的挂锁去锁任何东西。当然，拉维还会得到打开自己挂锁的钥匙。因为拉维所有的挂锁都一样，所有钥匙也都一样。到目前为止描述的场景显示在下页图中。我们将这一最初设置称为“实体挂锁把戏”（physical padlock trick）。

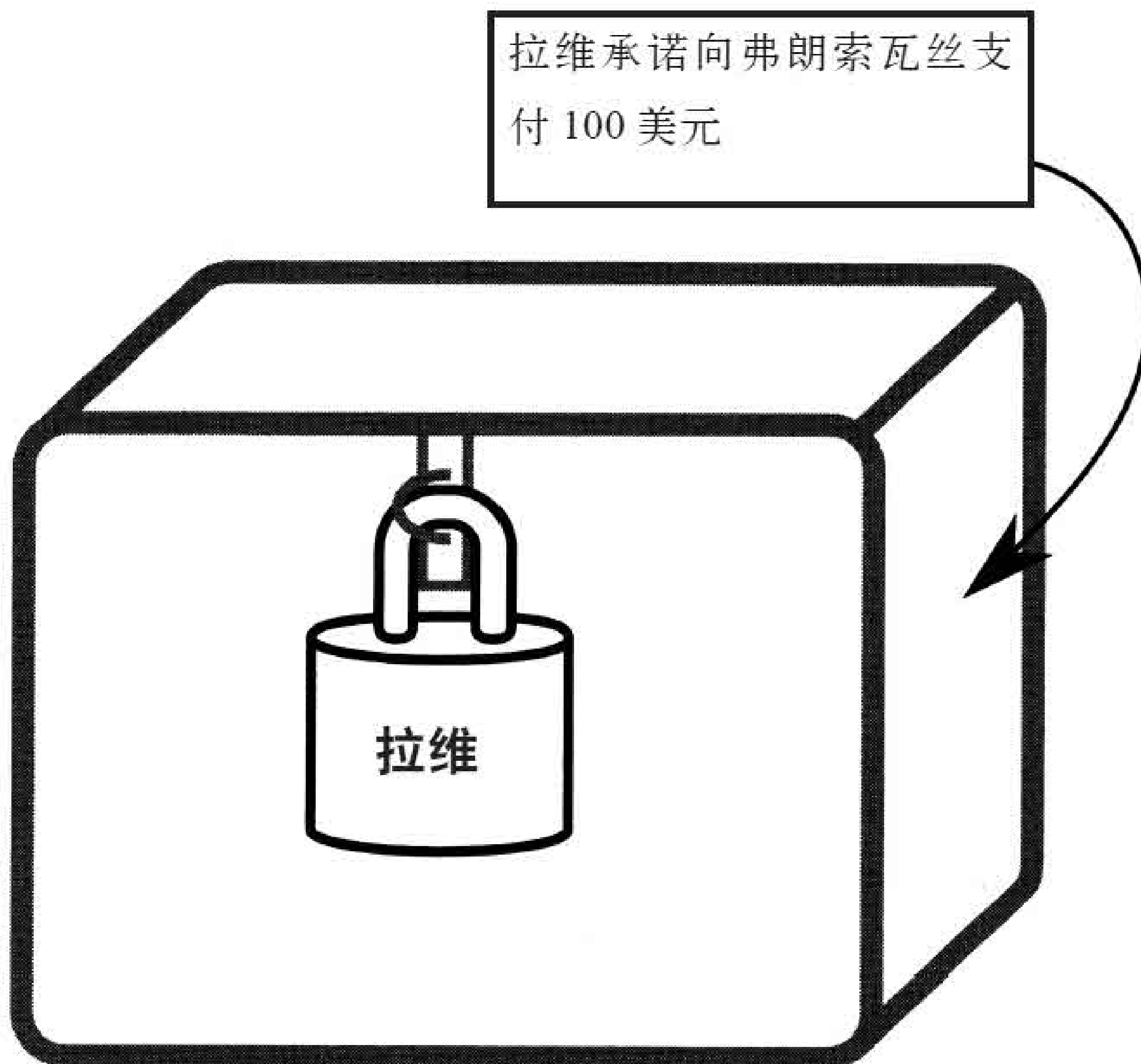
现在，让我们和之前一样假设，拉维欠弗朗索瓦丝100美元，而弗朗索瓦丝想用一种可验证的方法记录这一信息。换言之，弗朗索瓦丝想要和上图一样的文件，却又不必依赖于手写笔迹。下面就是这一把戏的运作方式。拉维制作一份文件，上面显示“拉维承诺向弗朗索瓦丝支付100美元”，但并不用在上面签名。拉维会制作一份该文件的拷贝，并将文件放在一个加锁的箱子内（加锁的箱子非常坚固，能用挂锁锁上）。最后，拉维用一个挂锁锁上箱子，并把锁上的箱子给弗朗索瓦丝。完整包裹如下图所示。准确地说，上锁的箱子就是文件的签名。注意，这对弗朗索瓦丝或一些受信目击者来说是个好主意，能观看签名的创建过程。否则，拉维就可以通过在箱内放上一份不同的文件来作弊。（可以论证的是，如果加锁的箱子透明，这一机制的效果会更好。毕竟，数字签名提供可靠性，而非隐秘性。不过，透明锁箱有点不符直觉，因此我们不会追求这一可能性。）




在实物挂锁把戏中，每位参与者都能得到独有的相同挂锁和钥匙。

也许你已经知道了弗朗索瓦丝验证拉维文件可靠性的方法了。如果任何人——甚至是拉维自己——试图否认这份文件的真实性，弗朗索瓦丝都能说“好，拉维，请借我一把你的钥匙。现在我要用你的钥匙来打开这个锁箱。”在拉维和其他目击者（甚至有可能是法院法官）的见证下，弗朗索瓦丝打开挂锁，展示锁箱里的内容。然后弗朗索瓦丝继续说：“拉维，因为你是唯一接触能用这把钥匙打开的锁的人，其他人不可能对锁箱内容负责。因此，是你，也只有你能写这张欠条，并把它放到锁箱内。你确实欠我100美元！”

尽管这一开始听起来很复杂，但这一认证方法既实用又强大。然而，这种方法确实有些缺点。其主要问题是，它要求拉维的配合：在弗朗索瓦丝能证明任何事情之前，她必须说服拉维借她钥匙。但拉维可以拒绝，或者更糟糕，假装合作却借给弗朗索瓦丝一把不同的钥匙——一把打不开拉维锁的钥匙。然后，当弗朗索瓦丝打不开锁箱时，拉维就能说：“看，这不是我的挂锁，伪造者可能制作了一份文件，并在我不知情的情况下把它放入锁箱中。”




为运用实物挂锁把戏制作一个可验证签名，拉维将一份文件拷贝放在锁箱内，并用一把他的挂锁锁上。

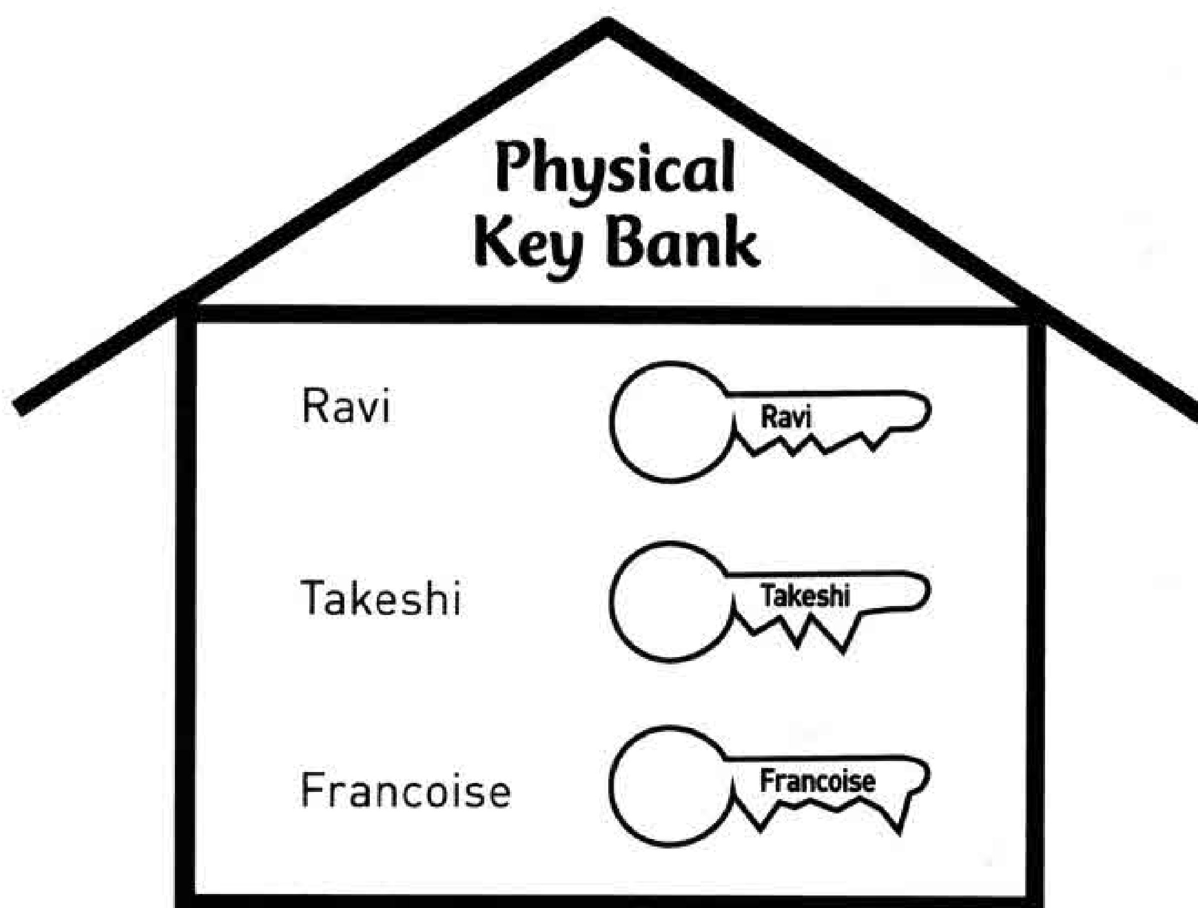
为防止拉维使用这种欺骗方法，我们仍需要依靠一个受信第三方，如银行。和前文的纸质签名银行相反，新银行会存储钥匙。因此，参与者不是给银行一份签名，而是给银行一把能打开自己挂锁的实物钥匙。下图显示了实物钥匙银行。

这个银行是这一谜题的最后部件，完整了实物挂锁把戏的解释。如果弗朗索瓦丝需要证明拉维写了欠条，她只要和一些目击证人把锁

箱带到银行，在银行用拉维的钥匙打开箱子即可。挂锁能打开就证明只有拉维能对箱中内容负责，而箱中正好是那份弗朗索瓦丝能试图验证的文件。

## 用乘法挂锁签名

结果证明，我们创建的钥匙—挂锁架构正是数字签名所需的方法。不过，很明显，我们不能在必须进行电子转换的签名上使用实物挂锁和实物钥匙。因此，下一步就是用类似的能数字化的数学对象取代挂锁和钥匙。具体来说，挂锁和钥匙将用数字代表，上锁或开锁动作将由钟算乘法（multiplication in clock arithmetic）代表。如果你对钟算不是很熟悉，请查看第四章的解释。




实物钥匙银行有很多钥匙，用于开启对应的挂锁。注意每把钥匙都是不同的。

为得到不能伪造的数字签名，计算机使用的钟大小非常大——钟大小长度基本在数十或数百位数。不过，在接下来的描述中，我们将使用的钟大小会非常小，基本不会在现实中运用，以确保计算能简单进行。

这一部分的所有例子都使用11作为钟大小。因为我们会多次用这个钟大小将数字乘起来。我在下面给出了一张表，列出了将小于11的数相乘得到的所有值。比如，让我们来手动计算7×5，不用那张表。我们先用普通算术计算答案：7 × 5 = 35。然后，我们保留用结果除以11后的余数。35除11得3（即33），余2。因此，最终结果是2。从表上看，第7行第5列的项的确是2。（你也能用第7列第5行——顺序并不重要，你可以自行查证。）你可以自行尝试另一对乘法例子，以确保自己理解。

在继续之前，我们需要略微变化一下我们尝试要解决的问题。在之前，我们一直在寻找让拉维“签署”一条给弗朗索瓦丝的消息（实际上是一张欠条）的方法。这条消息用日常英语写就。但从现在开始，只和数字打交道要方便得多。因此，我们必须承认，计算机将消息翻译成一个数字字符串让拉维签署很容易。之后，如果有人需要验证拉维对这个数字字符串的签名时，计算机逆转翻译并把数字转化成英语也很简单。在讨论校验和及更短符号把戏时，我们遇到了同样的问题。如果你愿意更细致地理解这一问题，请回顾更短符号把戏的讨论——这张图给出了一个在字母和数字之间翻译的简单、详尽的例子。

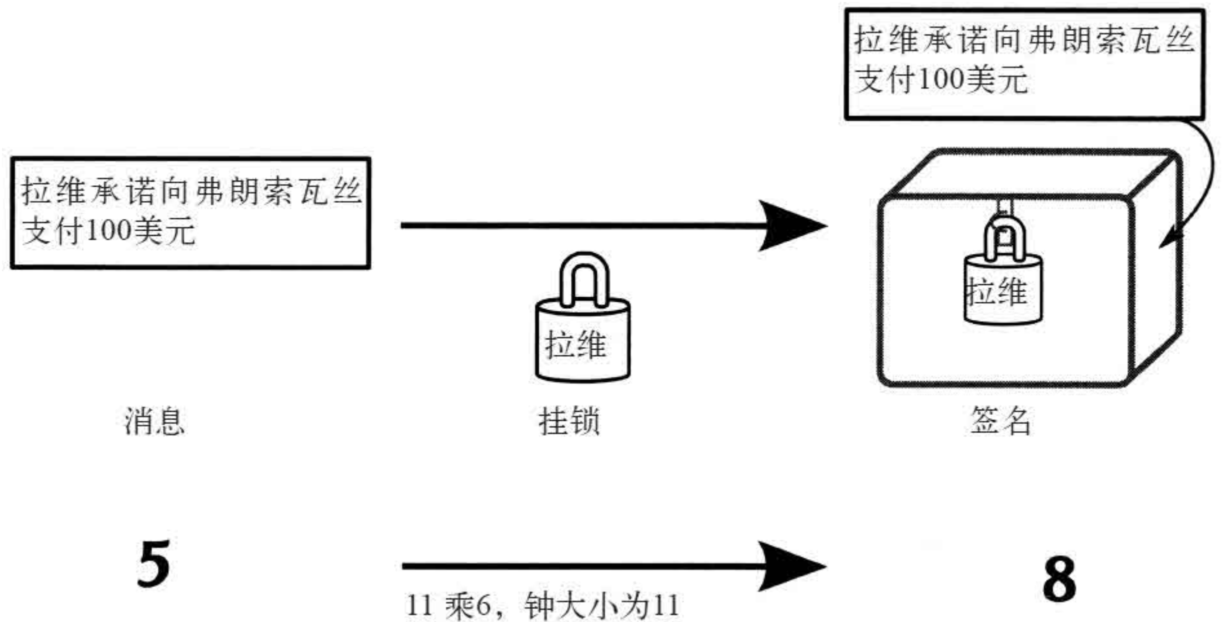
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| **2** | 2 | 4 | 6 | 8 | 10 | 1 | 3 | 5 | 7 | 9 |
| **3** | 3 | 6 | 9 | 1 | 4 | 7 | 10 | 2 | 5 | 8 |
| **4** | 4 | 8 | 1 | 5 | 9 | 2 | 6 | 10 | 3 | 7 |
| **5** | 5 | 10 | 4 | 9 | 3 | 8 | 2 | 7 | 1 | 6 |
| **6** | 6 | 1 | 7 | 2 | 8 | 3 | 9 | 4 | 10 | 5 |
| **7** | 7 | 3 | 10 | 6 | 2 | 9 | 5 | 1 | 8 | 4 |
| **8** | 8 | 5 | 2 | 10 | 7 | 4 | 1 | 9 | 6 | 3 |
| **9** | 9 | 7 | 5 | 3 | 1 | 10 | 8 | 6 | 4 | 2 |
| **10** | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |

钟大小为11的乘法表。

因此，拉维不会签署一条用英语写成的消息，而是必须签署一个数字字符串，如“494138167543……83271696129149”。不过，为简单起见，我们一开始会假设签署的消息非常之短：事实上，拉维的消息将由“8”或“5”这样的单个数字组成。不要担心：我们最终将学会签署长度更合乎情理的消息的方法。至于现在，最好还是坚持用单个数字消息。

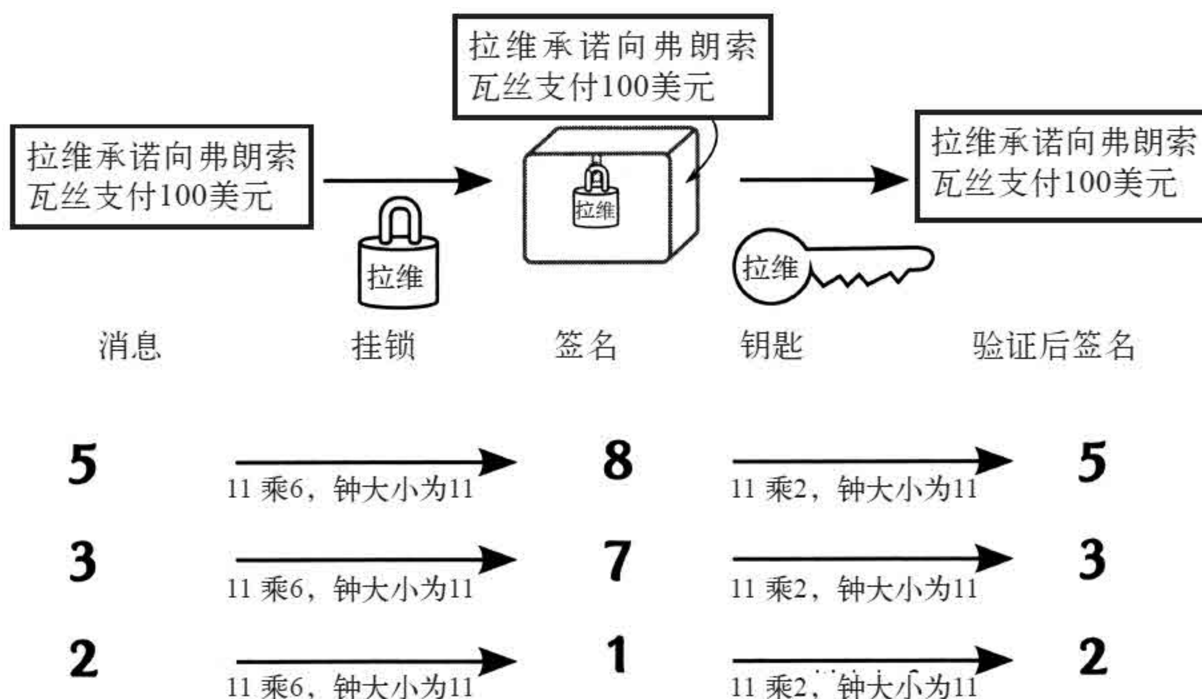
做好了这些准备之后，我们就准备好理解“乘法挂锁把戏”的核心了。和实物挂锁把戏一样，拉维会需要一把挂锁和一把打开挂锁的钥

匙。获得挂锁非常容易：拉维首先选择一个钟大小，然后选择比钟大小小的任意数作为数字“挂锁”。（实际上，一些数比其他数效果更好，但谈这些细节会离题太远。）为让例子具体化，假设拉维选择11作为钟大小，选择6作为挂锁。




如何制作一个数字签名，用一把“挂锁”“锁住”一条数字消息。图中最上面一行显示了如 何用一把实物挂锁在一个箱子中锁住一条消息。图中最下面一行显示了类似的数学操作，里面的消息是个数字（5），挂锁是另一个数字（6），上锁过程是已知钟大小的乘法。最终结果（8）就是消息的数字签名。

现在，拉维如何用挂锁把消息“锁”入锁箱呢？听起来也许很奇怪，拉维将用乘法做到这一点：拉维的“上锁”消息将由挂锁和消息相乘（钟大小为11）。记住，我们现在在处理一条由单个数字组成的消息。因此，假设拉维的消息是“5”。那么他的“上锁”消息会是6 × 5,通过钟算得到结果为8。（你可以用前页的乘法表进行双重核对）这一过程在上图中得到了总结。最终结果“8”就是拉维给原始消息的数字签名。




如何使用数字挂锁和对应数字钥匙“上锁”并在随后“解锁”一条消息。图中最上面一行显 示了实物版上锁和解锁。图中最下面三行显示了使用乘法给消息进行数字上锁和解锁的例子。注意，上锁过程得到了一个数字签名，而解锁过程则得到了一条消息。如果解锁消息与原始消息匹配，数字签名就得到了验证，原始消息为真。

当然，如果我们随后不能运用某种数学“钥匙”解开消息，这种数学“挂锁”也会毫无用处。幸运的是，有一种解锁消息的简单方法。这种方法是再次使用乘法（和之前一样应用钟大小），但这次我们会乘以一个不同的数——这个数字经过特别选取，能解开之前选取的挂锁数字。

让我们继续使用这个具体例子，拉维仍然用11作为钟大小，用6作为挂锁数字。结果显示，其对应的钥匙是2。我们怎么知道？稍后再回答这个重要问题。现在，让我们继续进行更容易的任务，在别人告诉我们钥匙的数值时验证钥匙奏效。之前提到过，我们通过将上锁消息和钥匙相乘就能解开上锁消息。我们已经在上图中见过，当拉维将消息5用挂锁6锁上时，他得到了上锁消息（或数字签名）8。要解锁上锁消息，我们将8用钟算乘以钥匙2，得到结果5。就像魔术一样，我们得到了原始消息5！整个过程如上图所示，你还能看到其他两个例子：消

息“3”被锁上后变成“7”，用钥匙界开始重新变成“3”。类似的，“2”被锁上后变成“1”，但钥匙将其转化回“2”。

上图还解释了该如何验证数字签名。你只需用签署者的乘法钥匙解锁签名即可。如果解锁消息和原始消息匹配，签名就是真实的。否则，签名就肯定是伪造的。这一验证过程在下页图中进行了更细致的展示。在这张表中，我们继续使用11作为钟大小，但为显示我们目前使用的数字挂锁和钥匙没有特殊之处，我们在这里使用了多个不同的值。特别是，挂锁值为9，而相应钥匙的值为5。表中第一项的消息是“4”，签名是“3”。签名解开后结果为“4”，与原消息匹配，因此签名为本人所写。表中第二行列举了一个相似的例子，消息为“8”，签名为“6”。但表的最后一行显示了签名系伪造时的情况。表中第四项的消息还是“8”，但签名却是“7”。这个签名解锁后是“2”，与原消息不匹配。因此，签名系伪造。

| 消息 | 数字签名（对于真签名，让消息和挂锁值9相乘。对于假签名，选择一个随机数。） | 解锁签名（要解锁签名，和钥匙值5相乘。） | 与消息匹配? | 系伪造? |
|---|---|---|---|---|
| 4<br>8<br>8 | 3<br>6<br>7 | 4<br>8<br>2 | 是<br>是<br>否！ | 否<br>否<br>是！ |

如何侦测一个伪造的数字签名。这些例子使用的挂锁值为9，钥匙值为5。前两个签名为真，但第三个签名系伪造。

如果你回想实物钥匙和挂锁的例子，你应该会记得挂锁有防止其他人使用的生物传感器——否则伪造者就能用拉维的挂锁将任意消息所在箱子中，进而伪造这条消息的签名。同样的道理也适用于数字挂锁。拉维必须保持他的挂锁号不为人知。每次他签署一条消息，拉维都要同时展示消息和签名，但并非用于制作签名的挂锁数。

拉维选择的钟大小和数字钥匙呢？这些也必须保密吗？不是。拉维可以公开宣布自己选择的钟大小和钥匙值，有可能会通过网站发布它们，而无须为验证签名的机制妥协。如果拉维真的发布了自己的钟大小和钥匙值，任何人都能获取这些数字并验证他的签名。乍一看，这种方法似乎非常方便——但还是有些重要的细微之处需要强调。

数字钥匙银行

| 名称 | 钟大小 | 数字钥匙 |
| --- | --- | --- |
| Ravi | 11 | 2 |
| Takeshi | 41 | 35 |
| Francoise | 23 | 15 |

数字钥匙银行。该银行的作用并非为数字钥匙和钟大小保密。相反，银行是获取与任何个人相关的真正钥匙和钟大小的受信权威。银行可以自由地向任何有要求的人展示这些信息。

## 数字钥匙银行

比如，这个方法消除了对受信银行的需求吗?纸质签名技术和实物挂锁—钥匙技术都需要受信银行。并非如此：仍然需要一个像银行这

样的受信第三方。没有受信第三方，拉维就能分发一个假钥匙值，让他的签名失效。更糟的是，拉维的敌人们可以制造一个新的数字挂锁和相对应的数字钥匙，并在一个网站上宣称这就是拉维的钥匙，然后用他们新伪造的数字挂锁签署任何他们想要的消息。任何相信这些新钥匙属于拉维的人都会相信，敌人的消息是拉维签署的消息。因此，银行的角色并不是为拉维的钥匙和钟大小保密。相反，银行是一个拉维的数字钥匙和钟大小值的受信权威。上图展示了这一情况。

总结这一讨论的一种有用方法是：数字挂锁是私有的，而数字钥匙和钟大小则是公开的。要承认的是，让钥匙“公开”有点违背常识，因为在我们的日常生活中，我们习惯于非常小心地保护我们的实物钥匙。为澄清钥匙这一不同寻常的用途，回想早先描述的实物挂锁把戏。在那个例子里，银行保有一把拉维的钥匙，并且很乐意将之借给任何想要验证拉维签名的人。因此，在某些意义上，实物钥匙就是“公开的”。同样的道理也适用于乘法钥匙。

现在是强调一个重要现实问题的好时机：假如我们想要签署不止一位数的消息呢？这一问题有多种答案。第一个解决方案是使用一个很大的钟大小：比如，如果我们使用一个100位数的钟大小，那么通过相同的方法，我们能用100位数的签名签署100位数的消息。对于长于100位数的消息，我们可以将消息分成100位数的块，并单独签署每个块。但计算机科学家们有一种更好的方法来做到这一点。实际情况是，出于签署的目的，通过应用一种名为加密哈希函数的转化方法，长消息能缩减为单个块（比如100位数）。我们已经在第五章谈到过加密哈希函数，它们在第五章被用作校验和，以确保大消息（如一个软件包）内容正确。这里的想法也很类似：一条长消息在签署前被缩减为小得多的块。这意味着极大的“消息”——如软件包——能高效签署。为简单起见，我们将在余下的章节中忽略长消息问题。

另外一个重要问题是：这些数字挂锁和钥匙最初来自哪里？之前曾提到，参与者们基本上能为挂锁选择任意值。不幸的是，隐藏在单词“essentially”（基本上）背后的细节1需要对数字理论有相当于本科课程的认识。但假设你没有机会学习数字理论，让我来讲述下列难题：如果钟大小是个素数，那么任何钟大小之外的正值都可以作为挂锁。否则，情况就要复杂得多了。素数没有除1和自身之外的商。本章中使用的钟大小11就是素数。

因此，选择挂锁很简单，特别是在钟大小为素数的情况下。但一旦选定挂锁，我们还需要应付解开选中挂锁的对应数字钥匙。这是个很有趣——也非常古老——的数学问题。实际上，这个问题的解决方案流传了数个世纪，而其中心思想甚至要更为古老：这一广为人知的技巧名为欧几里得算法，由希腊数学家欧几里得于2 000多年前发明。不过，我们不必在此追寻钥匙生成的细节，只需要知道，给出一个挂锁值，计算机就能通过一种名为欧几里得算法的知名数学技巧得出相对应的钥匙值。

如果你仍对这一解释不满意，也许在我稍后揭示戏剧性转折后，你会变得高兴。整个挂锁和钥匙的“乘法”方法有个根本缺陷，必须被抛弃。在下一部分，我们将用一种不同的方法处理挂锁和钥匙，一种实际上已经在现实中运用的方法。那么，为什么我还要解释有缺陷的乘法系统呢？主要原因是，所有人都熟悉乘法，这也意味着不需要一次性接受大量新知识就能解释系统。另外一个原因是，在有缺陷的乘法方法和我们将在下面考虑的正确方法之间，存在一些迷人的联系。

但在继续往下讲之前，让我们尝试理解乘法方法的缺陷。之前说过，挂锁值是私有的（或秘密的），而钥匙值则是公开的。刚刚也讨论过，签名机制的参与者可以自由选择钟大小（公开）和挂锁值（仍旧保密），然后通过计算机生成对应钥匙值（在这个乘法钥匙的特殊例子中，我们会使用欧几里得算法）。钥匙会存储在一个受信银行

中，而银行会向任何请求的人展示钥匙值。乘法方法的问题在于，用于从挂锁生成钥匙的同样把戏——基本上是指欧几里得算法——能非常完美地逆向运行：同样的技术能让计算机生成已有钥匙值对应的挂锁值！我们立马就知道了抛弃整个数字签名机制的原因。因为钥匙值是公开的，理应保密的挂锁值也能被任何人计算出来。而一旦你知道了某人的挂锁值，你就能伪造那个人的数字签名。

## 用指数挂锁签名

在这一部分，我们将升级有缺陷的乘法系统，代之以一种在现实中运用的著名数字签名机制RSA。新系统将使用一种不知名的操作求幂（exponentiation）取代乘法操作。事实上，我们经历了和第四章理解公钥加密相同的求幂步骤：我们先弄清一个使用乘法的简单的缺陷系统，然后研究使用求幂的真实系统。

因此，如果你对$5^9$和$3^4$等幂符号不熟悉，这是个重温“现实生活中的颜料混合把戏”内容的好时机。但是要提醒一句，$3^4$（“3的4次方”）意味着3×3×3×3。另外，我们需要几个技术术语。在如$3^4$这样的表述中，4被称作指数（exponent）或幂（power），而3则被称为底（base）。在一个底上应用指数的过程被称为“自乘幂次”（raising to a power），更正式的说法是求幂（exponentiation）。和第四章一样，我们将结合求幂与钟算。本章这一部分的所有例子都用22作为钟大小。我们唯一需要的指数是3和7，因此在下面提供了一张表，显示$n^3$和$n^7$的值，n为1到20的整数（钟大小为22）。

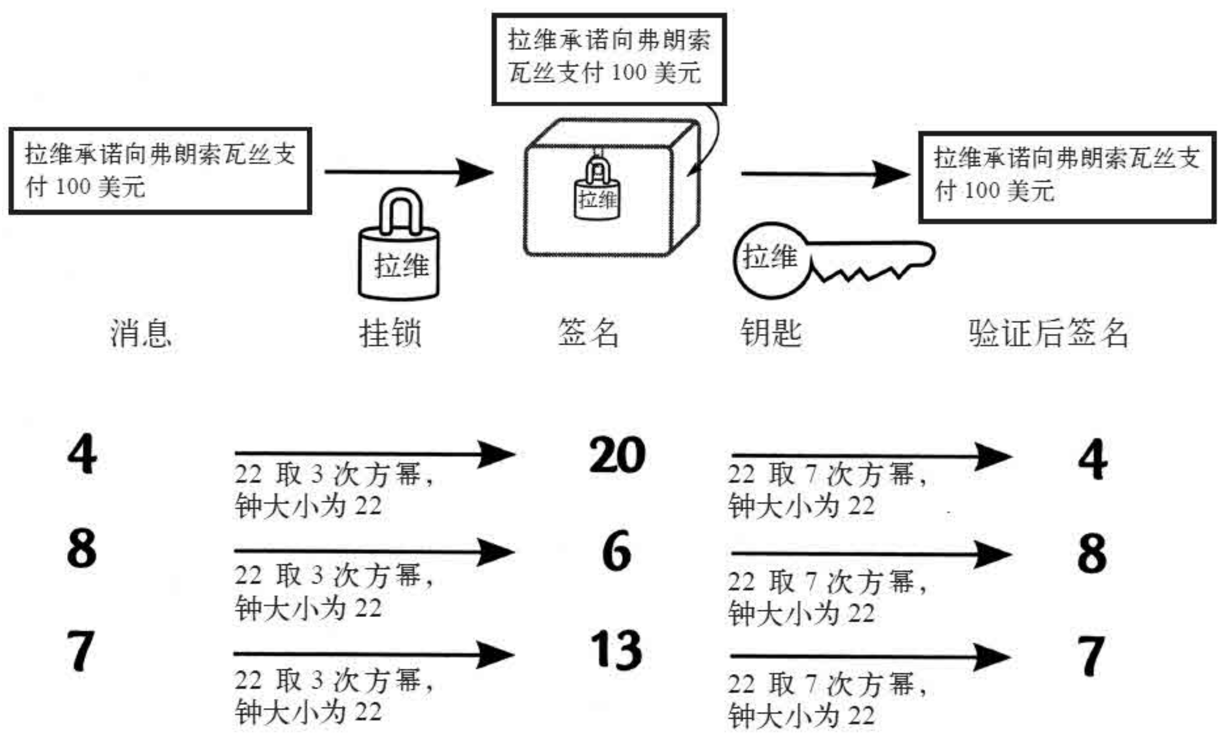
| $n$ | $n^3$ | $n^7$ |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 8 | 18 |
| 3 | 5 | 9 |
| 4 | 20 | 16 |
| 5 | 15 | 3 |
| 6 | 18 | 8 |
| 7 | 13 | 17 |
| 8 | 6 | 2 |
| 9 | 3 | 15 |
| 10 | 10 | 10 |
| 11 | 11 | 11 |
| 12 | 12 | 12 |
| 13 | 19 | 7 |
| 14 | 16 | 20 |
| 15 | 9 | 5 |
| 16 | 4 | 14 |
| 17 | 7 | 19 |
| 18 | 2 | 6 |
| 19 | 17 | 13 |
| 20 | 14 | 4 |

钟大小为22时n的三次方和七次方的值。

现在让我们核实表中的两个项，确保它们有意义。看一下对应n=4的行。如果我们不使用钟算，结果就是$4^3$ =4×4×4 = 64。但应用钟大小22后，64中减去2个22（44）余20。这是$n^3$栏那一项为20的由来。类似的，在不应用钟算的情况下，你能算出47 = 16384，比最接近的22的倍数正好大16（如果你感兴趣，22 × 744 = 16 368）。这解释了$n^7$栏那一项为16的由来。

现在我们最终准备好了观察实际运作中的数字签名。系统运作的方式和前一部分的乘法方法一样，只有一个例外：和用乘法给消息上锁及解锁不同的是，我们用求幂来进行这些操作。和前面的例子一样，拉维首先选择并公开钟大小。在这里，拉维用22作钟大小。然后他选择一个秘密挂锁值，这个值可以是钟大小之外的任意数（受制于

一些细节，我们将在稍后进行简短讨论）。在这个例子中，拉维选择3作为挂锁值。然后，他用计算机算出挂锁和钟大小对应的钥匙值。我们稍后会多了解一些与此有关的细节。但唯一重要的事实是，计算机能使用众所周知的数学技术，轻易根据挂锁和钟大小计算出钥匙。在这个例子中，结果证明钥匙值7对应先前选取的挂锁值3。




用求幂给消息上锁和解锁。

上图给出了一些拉维如何签署消息，以及其他人如何解锁并核实签名的具体例子。如果消息是“4”，签名是“20”：我们通过以挂锁值为幂，让消息求此幂得出结果。因此，我们要计算$4^3$，在应用钟大小之后得20。（不要忘记，你可以用上页的表轻易地核实这些计算。）现在，当弗朗索瓦丝想要验证拉维的数字签名“20”时，她先去银行得到拉维钟大小和钥匙的权威值。（银行和之前例子中的一样，除了数字不同外）然后弗朗索瓦丝用签名求钥匙值的幂，再应用钟大小：计算结果为$20^7 = 4$，你可以和上页图中的结果对照。如果结果和原消息匹

配（在这个例子中也的确匹配），签名为真。上图显示了消息“8”和“7”的相似计算。

下表再次展示了这一过程，这次强调了签名的验证过程。图中前两个例子和上一张图中的一样（分别是消息“4”和“8”），都是真签名。第三个例子的消息是“8”，签名是“9”。在应用钟大小和钥匙值解锁后，97的结果是15，和原消息不匹配。因此，第三条消息系伪造。

| 消息 | 数字签名（对于真签名，让消息求挂锁值3的幂。对于假签名，选择一个随机数。） | 解锁签名（要解锁签名，求钥匙值7的幂。） | 与消息匹配? | 系伪造? |
|---|---|---|---|---|
| 4<br>8<br>8 | 20<br>6<br>9 | 4<br>8<br>15 | 是<br>是<br>否！ | 否<br>否<br>是！ |

如何用求幂侦测一个伪造的数字签名。这些例子使用的挂锁值为3，钥匙值为7，钟大小为22。前两个签名为真，但第三个签名系伪造。

如前所述，这种求挂锁值的幂和求钥匙值的幂以RSA数字签名机制而闻名。这个名字由于20世纪70年代首次发表该系统的三位发明者姓的首字母组合而成：罗纳德·李维斯特（Ronald Rivest）、阿迪·沙米尔（Adi Shamir）和雷奥纳德·阿德尔曼（Leonard Adlemen）。RSA听起来可能很耳熟，因为我们已经在讨论公钥加密的第四章遇到过这个首字母缩略词。事实上，RSA既是一种公钥加密机制，又是一种数字签名机制——这绝非偶然，这两种算法之间有着很深的理论关系。在本章，我们只探索了RSA的数字签名面，但你可能已经注意到了一些和第四章思想惊人的相似之处。

如何在RSA系统中选择钟大小、挂锁和钥匙的细节非常吸引人，但在理解大致方法上却不需要这些细节。最重要的是，在这一系统中，一旦选定挂锁值，参与者就能轻易计算出合适的钥匙值。但其他任何人想要逆转这一过程都不可能：如果你知道某人正在用的钥匙和

钟大小，你也不能算出对应的挂锁值。这弥补了早前解释的乘法系统的缺陷。

至少计算机科学家认为RSA很安全，但没人能肯定。RSA是否真的安全？这个问题是整个计算机科学中最迷人、最令人烦恼的问题之一。要提一下，这个问题同时取决于一个未解决的古老数学问题，以及一个位于物理学和计算机科学研究交叉地带的最新热门主题。数学问题就是知名的整数分解（integer factorization）；而热门研究主题就是量子计算（quantum computing）。我们将逐一研究RSA安全性的这两个方面，但在这么做之前，我们需要更好地了解一下，像RSA这样的数字签名机制“安全”的真正含义是什么。

## RSA的安全性

所有数字签名机制的安全性都要归结到一个问题：“敌人能伪造我的签名吗？”对于RSA而言，这个问题可以转化为“敌人能根据我的公开钟大小和钥匙值计算出我的私人挂锁值吗？”“是的！”这一问题的简单答案也许会让你感到沮丧。事实上，你已经知道：通过试错总有可能算出某人的挂锁值。毕竟，我们有消息、钟大小和数字签名。我们知道挂锁值要小于钟大小，因此我们可以简单地逐一尝试所有可能的挂锁值，直到找到一个能生成正确签名的挂锁值。这就是一个消息求每个尝试挂锁值的幂的问题。在实际中，诀窍是RSA机制使用绝对大的钟大小——比如数千位数长。这样，即便使用现存最快的超级计算机，也要花数万亿年才能尝试所有可能的挂锁值。因此，我们对敌人是否能用某种方法计算出挂锁值不感兴趣。相反，我们想知道敌人是否能足够高效地这么做，从而造成实际威胁。如果敌人的最佳攻击方法就是试错——计算机科学家通常称之为暴力破解（brute force）——我们可以一直都选用足够大的数作为钟大小，从而让攻击变得不切实

际。另一方面，如果敌人的技术效率比暴力破解快很多，我们就可能有麻烦了。

比如，说回乘法挂锁和钥匙机制，我们知道签署人可以选择一个挂锁值，然后通过使用欧几里得算法计算钥匙值。但缺陷是敌人无须依靠暴力破解就能逆转过程：结果证明，欧几里得算法也能根据钥匙值计算出挂锁值，而这一算法要比暴力破解高效得多。这也是乘法方法被认为不安全的原因。

## RSA和因式分解的联系

我之前承诺过，要展示RSA的安全性与一个名为整数分解的古老数学问题之间的联系。为理解这一联系，我们需要多知道一些与如何选择RSA钟大小有关的细节。

首先，回顾一下素数的定义：素数就是只有1和它自身两个因子的数。比如，31是素数，因为只有1×31才能得到31。但33却不是素数，因为33=3×11。

现在，我们准备好研究签署人——如老朋友拉维——生成RSA钟大小的全过程。拉维做的第一件事是选择两个非常大的素数。基本上这些素数会有数百位数长，但和之前一样，我们用小例子代替。假设拉维选择了素数2和11，然后将两数相乘，得到钟大小2×11=22。钟大小会和拉维选取的钥匙值一道公开。但——这是关键时刻——钟大小的两个素数因子仍然保密，只有拉维知道。RSA之后的数学让拉维有方法使用这两个素数因子根据钥匙值算出挂锁值，反之亦然。

这一方法的细节在下图中得到了描述，但它们和我们的主要目的无关。我们需要知道的就是，拉维的敌人不能使用公开信息（钟大小和钥匙值）计算出他的保密挂锁值。但如果他的敌人也知道钟大小的

两个素数因子，他们就能轻易计算出保密的挂锁值。换言之，如果拉维的敌人能将钟大小因式分解，他们就能伪造拉维的签名。（当然，可能还有其他破解RSA的方法。对钟大小进行高效因式分解只是可能的攻击方法之一。）

在这个小例子中，将钟大小因式分解（并破解数字签名机制）非常容易：所有人都知道22=2×11。但当钟大小有数百位或数千位数长时，找出因子就变得极其困难了。事实上，尽管这个所谓的“整数分解”问题被研究了数个世纪，还没人找到一个足够高效的通用方法解决它，并对标准RSA钟大小造成危害。

数学史中充满了未解决问题，尽管这些迷人问题缺乏任何实际应用，单靠其美学特质就吸引了数学家进行深入研究。令人颇感惊讶的是，许多这类迷人但显然无用的问题后来都有了很大的实用价值——在一些例子中，这一价值只有在问题被研究数个世纪后才发现。

2 × 11 = 22

素数　素数　第一钟大小

↓减 1　↓减 1

1 × 10 = 10

第二钟大小

拉维选择两个素数（2和11）并将两数相乘得到钟大小（22）。我们将其称为“第一”钟大小，原因很快即会说明。接下

来，拉维将两个素数各减1，再将相减后得到的数相乘。这就得到了拉维“第二”钟大小。在这个例子中，拉维再将原素数各减1后分别得到1和10，因此第二钟大小为1×10=10。

这时，我们遇到了一个令人极其愉快的联系，这一联系和之前描述的有缺陷的乘法挂锁钥匙系统有关：拉维根据乘法系统选择了挂锁和钥匙，但却使用第二钟大小而非第一钟大小。结果显示，当使用第二钟大小10时，与之对应的乘法钥匙为7。我们能很快证明这一方式奏效：消息“8”乘上挂锁值为8×3=24，应用钟大小10后得“4”。用钥匙解锁“4”得4×7=28，应用钟大小后得“8”——和原消息一样。

现在拉维的工作完成了：他将乘法挂锁和刚刚选择的钥匙直接用作RSA系统中的指数挂锁和钥匙。当然，它们被当作指数使用时会选择第一钟大小22。

生成RSA钟、挂锁和钥匙值的所有细节。

整数分解这一问题由来已久。对其最早的严肃研究似乎是在17世纪，由数学家费马（Fermat）和梅森（Mersenne）进行。欧拉（Euler）和高斯（Gauss）——两位数学泰斗——也在接下来的世纪里对这一问题做出了贡献，其他许多人也贡献了自己的力量。但直到公钥加密于20世纪70年代发明，分解大数字的困难才成为一个实际应用的关键。你现在应该知道，任何发明一种高效分解大数字算法的人都能随意伪造数字签名。

在这听起来有点危言耸听之前，我要澄清一点，20世纪70年代后发明了无数其他数字签名机制。尽管每种机制都依赖于一些基本数学难题的难解度，但不同的机制依赖不同的数学难题。因此，发明一种高效的分解因子算法只会破坏类RSA机制。

另一方面，计算机科学家都对一个适用于所有这些系统的迷人问题感到困扰：没有一种机制被证明是安全的。每一种机制都依赖于一些很复杂、解题时间很长的数学难题。而在每个难题中，理论学家又不能证明没有高效解决方案存在。因此，尽管专家们认为可能性非常低，但原则上任何时候任何一种加密或数字签名机制都可能被攻破。

## RSA和量子计算机的联系

我兑现了揭露RSA和一个古老数学问题之间联系的承诺，但还未解释RSA与量子计算这一热门研究主题之间的联系。要探究这一联系，我们必须首先接受下列基本事实：在量子力学中，物体移动由概率主导——与经典物理学的决定论定律（deterministic laws）相反。因此，假如你用易受量子力学影响的部件搭建一台计算机，它所计算的值会由概率决定，而非经典计算机生成的由0和1组成的绝对确定的序列。看待这一情况的另一种角度是，量子计算机会同时存储多个不同的值：不同的值有不同的概率，但在你强迫计算机输出一个最终答案前，所有值都同时存在。这让量子计算机同时计算多个不同可能的答案成为可能。对于一些特定种类的问题，你可以使用“暴力破解”方法同时尝试所有可能的解决方案！

这确实只对特定种类的问题奏效，但整数分解恰巧就是量子计算机在执行效率上比经典计算机快得多的任务之一。因此，如果你能搭建一台能处理数千位数数字的量子计算机，你就能像先前解释的一样伪造RSA签名：因式分解公开钟大小，使用因子得到第二钟大小，再使用第二钟大小从公开钥匙值中获得私密挂锁值。

在我于2011年写下这些文字时，量子计算的理论超前实践很多。研究人员们尝试搭建真正的量子计算机，但目前为止由量子计算机执行的最大因式分解是15=3×5——离因式分解数千位数长的RSA钟大小

还相差甚远！而且在创造更大的量子计算机前，还有非常具体的问题需要解决。因此，没人知道量子计算机何时——或能否——大到能够一次性破解RSA系统。

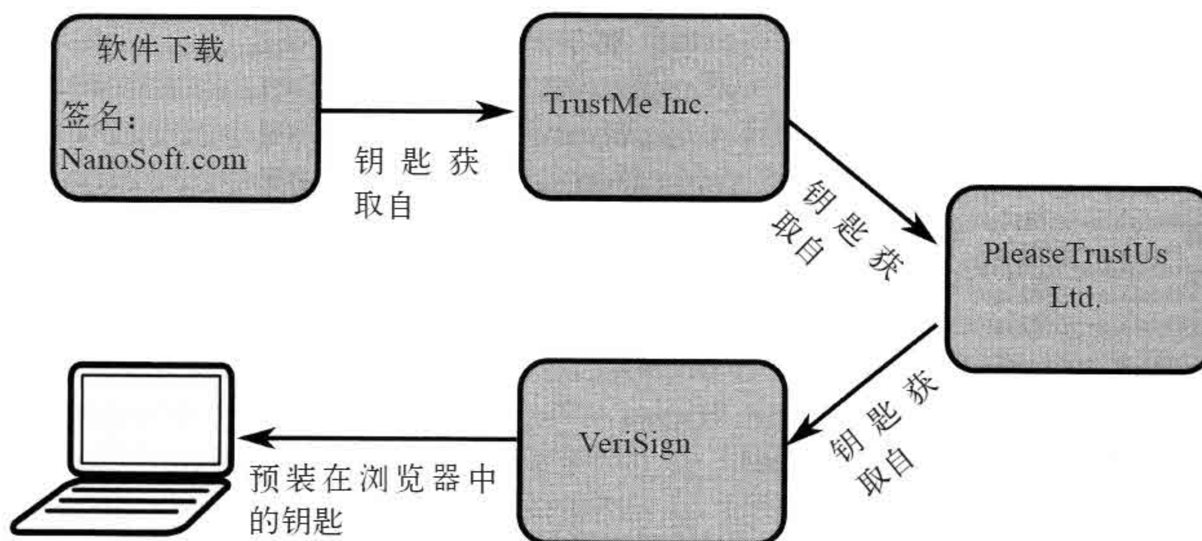
## 实际中的数字签名

在本章开头，我们了解到如你我的终端用户并没有太多数字签名的需求。一些计算机深度用户的确会对电子邮件消息等东西签名，但对于绝大多数人而言，数字签名的主要用途是验证下载后的内容。最明显的例子是，当你下载一份新软件时，如果软件有签名，计算机就会使用签名者的公钥“解开”签名，并将结果和签名者的“消息”——也就是软件本身——对比。[之前提过，在实践中，软件会在签名前被缩减成一条小得多的名为安全哈希（secure hash）的消息。]如果解开的签名与软件匹配，你会看到一条鼓舞人的消息；否则，你会看到一个更加可怕的警告：这两个例子都在本章开头的图中有显示。

我一直都强调，本章所有机制都需要某种受信“银行”存储签名者的公钥和钟大小。幸运的是，你可能也注意到了，每次下完一些软件，你都无须去一家真正的银行。在现实生活中，存储公钥的受信组织被称为认证机构（certification authorities）。所有认证机构都有维护服务器，有需要的人可以连接这些服务器下载公钥信息。因此，当你的计算机收到一个数字签名时，签名还会伴有说明可以去哪些认证机构获取签名者公钥的信息。

你可能已经注意到了这个问题：的确，计算机能先一步通过制定认证机构验证签名，但我们怎么才能信任机构本身呢？我们的做法就是将验证一个组织（发送软件给你的组织，如NanoSoft.com）身份的问题，转移为验证另一个组织（认证机构，如TrustMe Inc）身份的问

题。不管你相不相信，这个问题基本上通过认证机构（TrustMe Inc.）的数字签名指引你前往另一个认证机构（如PleaseTrustUs Ltd.）验证。这种信任链可以无限扩展，但我们永远会卡在同一问题上：我们如何能信任链条末端的组织？答案如上图所示，一些特定组织被正式指认为所谓的根认证机构。VeriSign、GlobalSign和GeoTrust等都是较为知名的根认证机构。在你有要求时，众多根认证机构的联系细节（包括互联网地址和公钥）会预装到浏览器软件中，这也是数字证书的信任链稳立于一个值得信任的起点的原因。




获取验证数字签名所需要的信任链。

## 悖论解决

在本章开头，我指出“digital signature”（数字签名）这一短语可以被视为矛盾修饰法：任何数字的东西都能被拷贝，而签名不应该能被拷贝。如何解决这一悖论呢？答案是，一个数字签名同时依赖一个只有签名者知道的秘密和被签署的消息。对于由某一特别实体签署的所有消息，秘密（我们在全章中称其为挂锁）都相同。因此，任何人都能轻易拷贝签名的事实就变得无关紧要了：签名只可能转化成一条相同的消息，仅仅是拷贝签名并不构成伪造。

这一悖论的解决方案不仅仅是个精巧美丽的思想。数字签名还起着巨大的实际作用：没有数字签名，我们所知的互联网就不会存在。数据仍可以通过加密安全交换，但要验证接收数据的来源就要困难得多。这一伟大思想和如此广泛的实际影响相结合，无疑让数字签名成为计算机科学中最伟大的成就之一。

# 第十章　什么可以计算——有些程序不可能存在

让我提醒你一些计算机的问题。

——理查德·费曼（Richard Feynman），
1965年诺贝尔物理学奖获得者

我们已经见识过很多聪明、强大且精准的算法——这些算法将计算机的生硬金属化为你指尖的精灵。事实上，根据前面章节的描述，很自然让人想到：有什么事情是计算机不能为我们做到的吗？如果我们把范围限制在计算机现在能做的事情上，答案很清楚：目前，计算机在许多有用任务（大多数设计某种形式的人工智能）上表现不好。例如英语和汉语这些语言之间的高质量翻译；在繁忙的城市环境中自动控制车辆安全快速行驶；给学生功课打分（作为老师，这对我来说是个大问题）。

然而，正如我们已经看到那样，一个真正聪明的算法能取得的成果常常出人意料。也许就在明天，有人发明了一种完美驾驶汽车的算法，或一种在给学生打分上效果很好的算法。这些问题看起来的确很困难，但是否困难到不能解决呢？的确，有问题能难到永远没有人能发明解决它的算法吗？在本章，我们会看到答案是肯定的：有些问题计算机永远也解决不了。这一显著事实——一些事情“可计算”而其他事情则不能——与我们在前面章节中所见的许多算法形成有趣对比。不管未来会发明多少聪明算法，有些问题的答案永远也“不可计算”。

不可计算问题的存在本身就足够震撼，但发现它们的故事更令人印象深刻。在发明出第一台电子计算机前，这类问题的存在就已为人所知。有两位数学家——一位是美国人，一位是英国人——于20世纪30年代末各自独立发现了不可计算问题，比第一台真正的计算机自第二次世界大战之间出现早数年。那位美国人是阿隆佐·邱奇（Alonzo Church），其在计算理论上的突破性工作至今仍是计算机科学许多方面的基础。那位英国人只能是阿兰·图灵，其被普遍视为创建计算机科学最重要的人物。图灵的工作横跨计算思想的整个范围，从复杂的数学理论、伟大的哲学到大胆实际的工程学。在本章，我们追随邱奇和图灵的脚步，进行一次最终会展示某个特殊任务不可能使用计算机完成的旅程。这一旅程将于下一部分对漏洞和崩溃的讨论开始。

## 漏洞、崩溃及软件的可靠性

近年来计算机软件的可靠性得到了大幅提升，但我们都知道，假设软件会正确运行仍然不是一个好主意。最通常的情况是，即便高质量、编写良好的软件都会做些偏离其原有目的的事。最糟糕的情况是，软件"崩溃"，你丢失了正在处理的数据或文件（或你正在玩的视频游戏——非常令人不解，我就碰到过这种事）。但任何在20世纪80年代和90年代见过家用计算机的人都能作证，当时计算机程序崩溃的频率要比21世纪大得多。取得这一提升有许多原因，但主要原因是自动化软件检查工具上取得的巨大进步。换言之，一旦一组计算机程序员编写完一个复杂的大型计算机程序，他们就能使用一个自动工具检查可能导致这一新建软件崩溃的问题。而这些自动化检查工具在发现潜在错误上也变得越来越好。

这自然会让人思考一个问题：自动化软件检查工具能否发展到可以侦测所有计算机程序中所有潜在问题的地步呢？要是能这样肯定很好，因为这能一劳永逸地消除软件崩溃的可能性。我们将在本章了解

到，永远也达不到这种软件理想境界：可以证明不可能有软件检查工具能侦测出所有程序中所有可能的崩溃。

在这里，值得花多点时间解释一些事情“可以证明不可能”的意思。在物理学和生物学等大多数科学中，科学家们会对特定系统的行为方式提出假说，并开展实验，验证假说是否正确。但由于实验本身有一定的不确定性，并不能百分百确定假说正确与否，即便实验本身非常成功。然而，与自然科学形成鲜明对比的是，对数学和计算机科学的一些结果是可能百分百予以确定的。只要你接受数学基本定理（如1+1=2），数学家使用的演绎推理链可以完全确定其他众多语句为真（比如，“任何以5结束的数都能被5相除”）。这种推理并不涉及计算机：数学家只使用一支铅笔和一张纸就能证明无可争辩的事实。

因此，在计算机科学中，当我们说“X可以证明不可能”时，我们并不只是说X似乎非常困难，或在实际中也许不可能实现，而是百分百确定X不可能实现，因为有人用演绎数学推理链证明了这一点。举个简单例子，“10的倍数以数字3结尾可以证明不可能”。另一个例子是本章的最终结论：可以证明不可能存在一个能侦测所有计算机程序中所有潜在崩溃的自动化软件检查器。

## 证明一些事情不为真

要证明这种崩溃侦测程序不可能存在，我们将要使用一种被数学家称为反证法（proof by contradiction）的技巧证明。尽管数学家喜欢对这一技巧宣示权利，但实际上人们在日常生活中常常用到它，经常连想都没想过。举个简单例子：

一开始，我们要同意下面两个事实，即便是最持怀疑的历史学家也不会对此有异议。

1. 美国内战发生在19世纪60年代。

2. 亚伯拉罕·林肯是美国内战期间的总统。

现在，假设我作出以下声明："亚伯拉罕·林肯生于1520年。"这一声明是真是假？即便你除了上面两个事实外对亚伯拉罕·林肯一无所知，你怎能快速判断我的声明为假呢？

最有可能的情况是，你的大脑会经历类似下面的推理链：（1）没有人的寿命能超过150年，因此如果亚伯拉罕·林肯生于1520年，他最迟也是在1670年死亡；（2）林肯是美国内战期间的总统，因此内战必须发生在林肯死亡前，也就是发生在1670年前；（3）但这不可能，因为所有人都认同美国内战发生在19世纪60年代；（4）因此，林肯不可能生于1520年。

让我们尝试更细致地检验这一推理。为何得出最开始声明为假的结论有效？这是因为我们证明这一断言与一些已知为真的事实矛盾。特别是，我们证明最初声明暗示美国内战发生在1670年前——这与美国内战发生在19世纪60年代的已知事实相矛盾。

反证法技巧极其重要，我还要再多举点数学例子。假设我做出下列断言："人的心脏10分钟内平均跳6 000次。"这一断言是真是假？你也许很快就会怀疑，但你要如何证明其为假呢？现在，在继续往下读前，花几秒钟分析下你的思考过程。

我们可以再用反证法。首先，出于论证的目的，假设这一断言为真：人的心脏10分钟平均跳约6 000次。如果这一断言为真，一分钟心脏会跳动多少次？平均来看，用6 000除以10，也就是一分钟600次。即便不是医学专家，你也知道这比正常心率高很多，正常心率约为每分钟50次至150次心跳。因此，原始声明与已知事实矛盾，肯定为假：人的心脏10分钟平均跳动约6 000次为假。

用更抽象的术语表示，可以如下总结反证法：假设怀疑某个声明S为假，但你却想确信无疑地证明其为假。首先，你假设S为真。通过进行一些推理，你得出某个声明T也必须为真。然而，如果已知T为假，就出现了矛盾。这证明你的原始假设（S）也必为假。

数学家可以更为简短地说明这些，只要说“S导出T，但T为假，因此S为假”这样的话即可。简言之，这就是反证法。下表展示了如何将这一

抽象反证法与上面两个例子联系起来：

| | 第一个例子 | 第二个例子 |
|---|---|---|
| S（原始声明） | 林肯生于1520年 | 人的心脏10分钟平均跳6 000次 |
| T（由S导出，但已知其为假） | 美国内战发生在1670年前 | 人的心脏每分钟跳600次 |
| 结论：S为假 | 林肯并非生于1520年 | 人的心脏10分钟内不会跳6 000次 |

到这里，我们进行反证法的旅程就完成了。本章的最终目标是通过反证证明，不存在一个能侦测其他程序中所有可能崩溃的程序。但在迈向这一最终目标前，我们需要熟悉一些和计算机程序有关的有趣概念。

## 分析其他程序的程序

计算机会严格按照计算机程序的指令行事。它们完全按照决定论方式行事，因此每次运行同一计算机程序都会得到相同的结果。输出是对还是错？事实上，我还没有给你足够信息来回答这一问题。的确，特定的简单计算机程序每次运行后都能得到相同的结果，但我们

每天使用的绝大多数程序每次在运行时看起来都非常不同。想想你最喜欢的文字处理程序：每次启动时屏幕看起来都一样吗？当然不是——屏幕出现什么取决于你打开的文件类型。如果我使用Microsoft Word软件打开文件“address-list.docx”，屏幕就会显示一个我保存在电脑上的地址列表。如果我用Microsoft Word软件打开文件“bank-letter.docx”，我就能看到昨天写给银行的信件文本。（如果你不认识“.docx”，请在下页的框中查找关于文件名后缀的内容。）

有件事我们要非常清楚：我在这两个例子中使用的都是同一款计算机程序，也就是Microsoft Word软件，只是每个例子中的输入不同罢了。不要被所有现代操作系统都能让你双击一个文件来运行一个计算机程序这一事实所愚弄。那只是你友好的计算机公司（最可能是苹果或微软）提供给你的一项便利。当你双击一份文件时，一个特定的计算机程序就会运行，程序会将文件当作输入使用。而程序的输出就是你在屏幕上看到的，这取决于你点击了什么文件。

在整章中，我都会使用如“abcd.txt”这样的文件名。英文句号后面的部分被称为文件名的“后缀”（extension）——“abcd.txt”的后缀就是“txt”。绝大多数操作系统用一个文件名的后缀判定文件包含什么类型的数据。比如，“.txt”文件通常包含纯文本，“.html”文件通常包含网页，而“.docx”文件则包含Microsoft Word文件。一些操作系统默认隐藏这些后缀，只有在操作系统中关闭“隐藏后缀”功能后才能看到。在网络中搜索“unhide file extensions”（显示文件后缀）会显示如何操作的指南。

一些关于文件名后缀的技术细节。

在现实中，计算机程序的输入和输出要比这复杂得多。比如，当你点击一个程序的菜单或打字时，你都在给程序以额外输入。当你保

存一个文档或其他文件时，程序会创造额外输出。但为保持简单起见，假设程序接收的都是同样的输入，即存储在你计算机上的一个文件。我们还假设程序得到相同的输出，即你显示屏上的图形窗口。

不幸的是，双击文件的现代便利产生了一个重要问题。操作系统使用多种聪明的把戏来推测，当你双击一个文件时，你希望运行哪个程序。但用任意程序打开任意文件是可能的，意识到这一点很重要。换言之，你可以将任意文件作为输入来运行任意程序。这一切是怎么做到的？下面的框列出了几种可以让你尝试的方法。这些方法不会在所有操作系统上奏效，也不能对所有输入文件奏效——不同的操作系统用不同的方式启动程序，有时候出于安全考虑，它们会限制输入文件的选择。不管怎样，我强烈建议你花几分钟在电脑上试验，以确信你钟爱的文字处理程序可以用多种不同的输入文件运行。

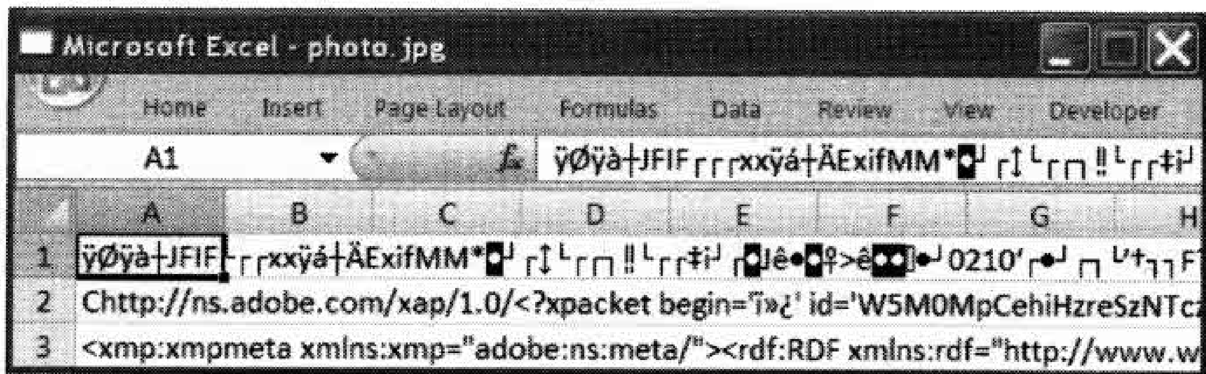
下面列举三种用stuff.txt作为输入文件运行Microsoft Word软件的方法：

·右键点击stuff.txt,选择“打开方式”，选择Microsoft Word。

·首先，用操作系统的功能在桌面上放置一个Microsoft Word的快捷方式。然后将stuff.txt拖动到这个Microsoft Word快捷方式中去。

·直接打开Microsoft Word软件，选择“文件”菜单的“打开”命令，确保选项显示“所有文件”，然后选择stuff.txt。

用某一特定文件作为输入运行一个程序的多种方法。




Microsoft Excel将“photo.jpg”作为输入运行的结果。输出虽是乱码，但重要的是，原则上你能用任何输入运行任意程序。

很显然，如果使用程序打开本非其能处理的文件，你会得到相当出人意料的结果。在上图中，你可以看到用电子表格程序Microsoft Excel打开图片文件“photo.jpg”时出现的情景。这个例子得到的输出是乱码，毫无用处。但电子表格程序的确运行了，并生成了一些输出。

上面的例子看起来已经很荒谬了，但我们要更加疯狂一点。你应该还记得，计算机程序本身就作为文件存储在计算机硬盘上。通常，这些程序的名字以“.exe”结尾，“exe”是“executable”（可执行）的缩写，表示你能“execute”（执行）或运行程序。因为计算机程序只是硬盘上的文件，我们可以将一个计算机程序作为另一个计算机程序的输入。比如，Microsoft Word程序在计算机上以文件“WINWORD.EXE”存储，通过将文件WINWORD.EXE作为输入来运行电子表格程序，我能得到如下图中所示的乱码。

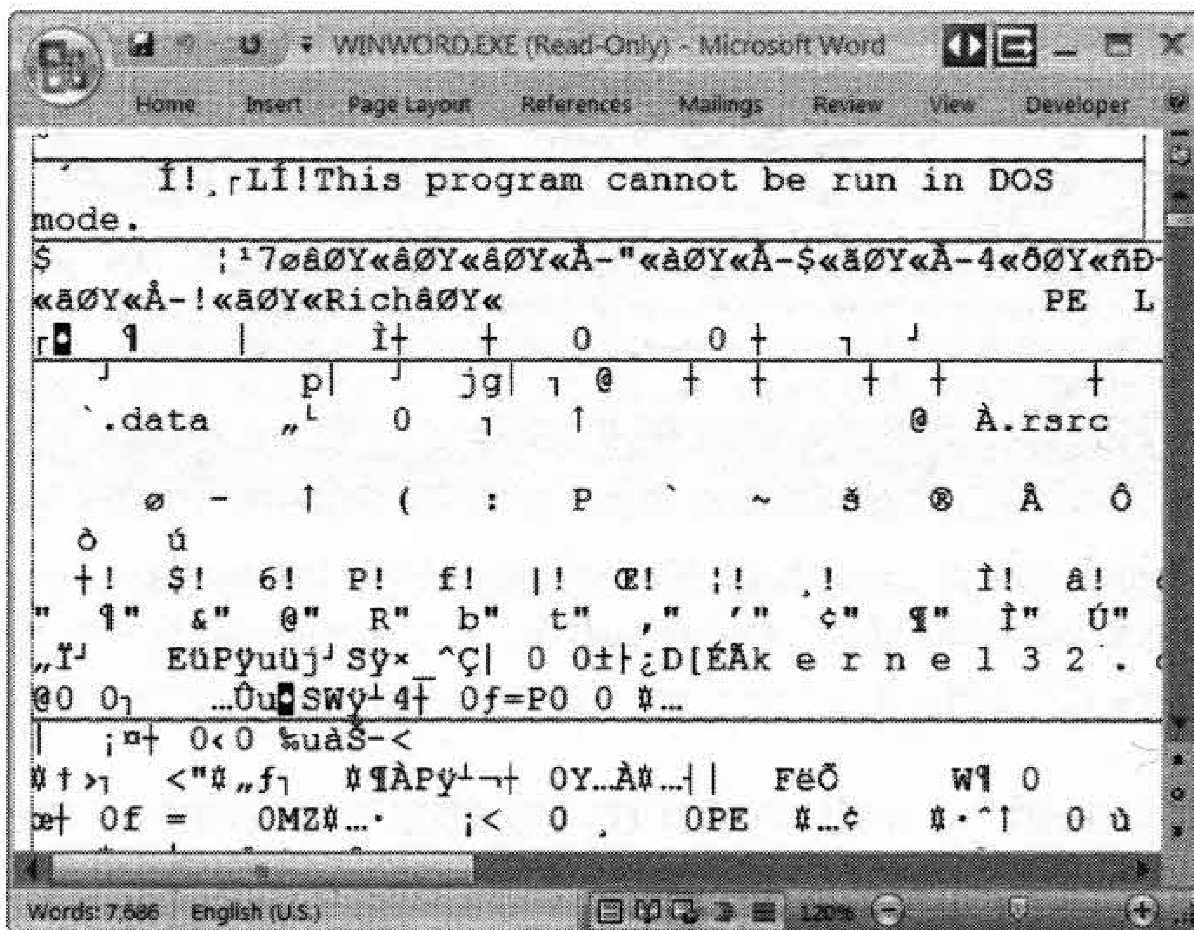
用Microsoft Excel检查Microsoft Word。当Microsoft Excel打开文件WINWORD.EXE，得到的结果是乱码一点也不让人意外。

和前面一样，值得亲自试一试这个实验。要进行这一实验，你需要先找到文件WINWORD.EXE。在我的计算机上，WINWORD.EXE位于文件夹“C:\Program Files\Microsoft Office\Office12”内，但其具体位置取决于你使用的操作系统以及安装的Microsoft Office版本。在查看这一文件夹之前，你也许还需要启用查看“隐藏文件”的功能。顺便提一句，你可以用任何电子表格或文字处理程序进行这一实验（以及下面的一个实验），尝试并不一定要用Microsoft Office。

我们还要做最后一件傻事。如果用一个计算机程序自身作为输入来运行这个程序呢？比如，如果我用文件WINWORD.EXE作为输入来运行Microsoft Word呢？尝试这一实验很容易。下图显示了我在计算机上试验的结果。和之前的几个例子一样，程序运行良好，但屏幕上的输出绝大多数是乱码。（自己尝试一下。）

那么，所有这些例子想表达什么？论述的目的是，让你能对一些在运行程序时出现的更难懂的事情见怪不怪。现在，你应该适应这三个略显奇怪但在后面非常重要的想法了。第一，任何程序可以将任何文件作为输入运行，但通常输出结果为乱码，除非输入文件本该配合

由你选择运行的程序。第二，我们发现，计算机程序作为文件存储在计算机磁盘上，因此一个程序可以用另一个程序作为其输入文件运行。第三，我们意识到，计算机程序能将其自身文件作为输入运行。到目前为止，第二种和第三种活动的结果一直是乱码，但我们将在下一部分看到一个这些把戏最终结出了一些果实的迷人例子。




Microsoft Word检查自身。打开的文档是文件WINWORD.EXE，即你点击Microsoft Word时实际运行的计算机程序。

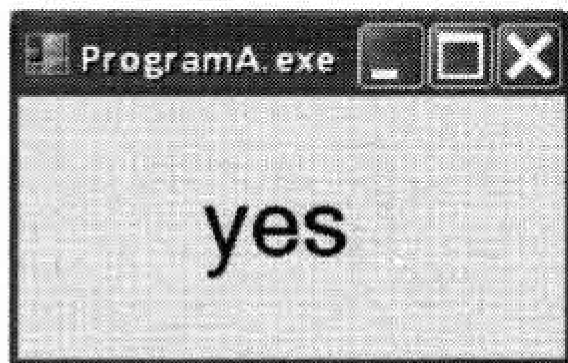
## 有些程序不可能存在

计算机在执行简单指令上非常棒——事实上，现代计算机每秒能执行数十亿次简单指令。你也许会想，任何任务都能以简单、精确的英语描述并写成一个计算机程序，由计算机执行。我在这一部分的目

标是说服你与之相反的情况为真：有些简单、精确的英语声明确实不可能写成计算机程序。

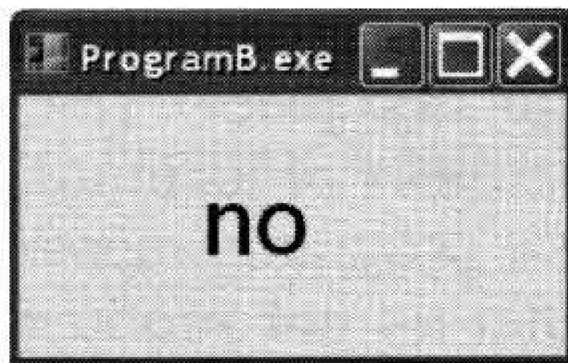
## 一些简单的是—否程序（Yes–No Program）

为让这一部分的内容尽可能简单，我们只会思考一种非常无聊的计算机程序。我们称它们为“是—否”程序，因为这些程序唯一能做的就是弹出一个对话框，对话框的内容要么是单词“yes”（是）或单词“no”（否）。比如，我在几分钟前写了一个名为ProgramA.exe的计算机程序，这个程序只能生成下列对话框：




注意看对话框的标题栏，你能看到生成这一结果的程序名ProgramA.exe。

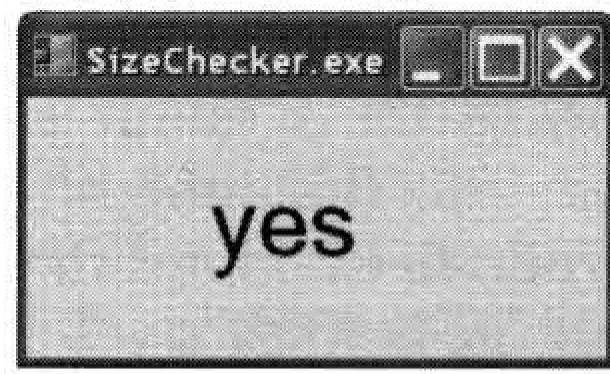
我还写了另一个名为ProgramB.exe的计算机程序，其输出为“no”而非“yes”。



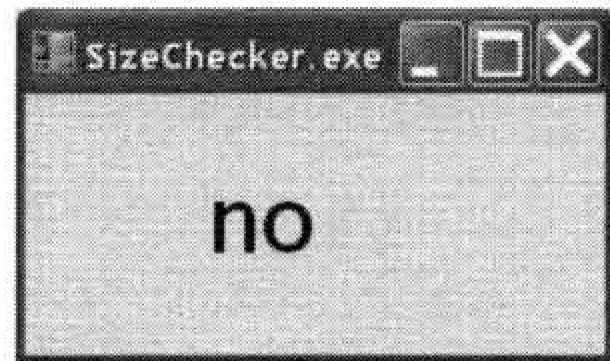

ProgramA和ProgramB都非常简单——事实上，简单到不需要任何输入。（如果它们确实接收到了输入，它们也会忽略。）换言之，它

们就是那些不管给予任何输入，每次运行都表现相同的程序的例子。

我还创造了一个名为SizeChecker.exe的程序，它是这些是—否程序中很有趣的例子。给该程序输入一个文件，如果文件大于10 KB，程序就输出“yes”，反之输出“no”。如果右键点击一个50 M的视频文件（假设是mymovie.mpg），选择“打开方式”，选取SizeChecker.exe，你就会看到如下输出：




另外，如果我在一条3KB的小电子邮件消息上运行同一程序（假设是myemail.msg），输出自然不同：




因此，SizeChecker.exe也是有时输出“yes”，有时输出“no”的是—否程序例子。

现在思考接下来这个稍有不同的程序，我们称之为NameSize.exe。这个程序会检验其输入文件的名称，如果文件名只有一个字母长，NameSize.exe输出“yes”；反之输出“no”。这个程序会有

哪些可能的输出？根据定义，所有输入文件的名称都至少有一个字母长（否则，文件就没有名称，也就不能在一开始选取它）。因此，NameSize.exe会一直输出“yes”，不管输入是什么。

顺便说一句，上面提到的几个程序是用其他程序作为输入时，不生成乱码的首批程序例子。比如，文件NameSize.exe的大小只有约8 KB。因此，如果你用NameSize.exe作为输入运行SizeChecker.exe，输出为“no”（因为NameSize.exe不超过10 KB）。我们甚至可以让SizeChecker.exe在自身上运行。这次会输出“yes”，因为SizeChecker.exe要大于10 KB——事实上，约为12 KB。类似的，我们可以让NameSize.exe用自身作为输入运行；由于文件名“NameSize.exe”包含不止一个字母，输出应该是“yes”。不得不承认，我们到目前为止讨论的所有是—否程序都相当无聊，但理解它们的行为很重要，请逐行查看下面的表，确保你赞同每项输出。

## AlwaysYes.exe：一个分析其他程序的是—否程序

| 运行的程序 | 输入文件 | 输出 |
|---|---|---|
| ProgramA.exe | address-list.docx | yes |
| ProgramA.exe | ProgramA.exe | yes |
| ProgramB.exe | address-list.docx | no |
| ProgramB.exe | ProgramA.exe | no |
| SizeChecker.exe | mymovie.mpg (50MB) | yes |
| SizeChecker.exe | myemail.msg (3KB) | no |
| SizeChecker.exe | NameSize.exe (8KB) | no |
| SizeChecker.exe | SizeChecker.exe (12KB) | yes |
| NameSize.exe | mymovie.mpg | yes |
| NameSize.exe | ProgramA.exe | yes |
| NameSize.exe | NameSize.exe | yes |

一些简单是—否程序的输出。注意无视其输入一直输出“yes”的程序（如ProgramA.exe、NameSize.exe）和有时输出“no”（如SizeChecker.exe）或一直输出“no”的程序（如ProgramB.exe）之间的区别。

现在我们要思考一些更有趣的是—否程序。我们将要研究的第一个程序名为“AlwaysYes.exe”。这个程序会检测输入文件，如果输入文件本身是一个永远输出“yes”的是—否程序，则AlwaysYes.exe会输出“yes”。反之，AlwaysYes.exe的输出为“no”。注意AlwaysYes.exe能完美运行任何种类的输入文件。如果你给AlwaysYes.exe的输入文件不是可执行程序（假设是addresslist.docx），程序会输出“no”。如果你给AlwaysYes.exe的输入文件是可执行程序（假设是WINWORD.EXE），

程序会输出“no”。如果你用一个是—否程序作为输入，但这个是—否程序有时会输出“no”，那么AlwaysYes.exe会输出“no”。只有在你输入一个不管输入是什么永远输出“yes”的是—否程序时，AlwaysYes.exe才会输出“yes”。到目前为止，我们已经在讨论中见了两个这样的程序例子：ProgramA.exe和NameSize.exe。下表总结了AlwaysYes.exe用多个不同输入文件时的输出，包括运行AlwaysYes.exe自身的情况。正如你在表中最后一行所看到的，AlwaysYes.exe在运行自身时输出“no”，因为AlwaysYes.exe在输入一些文件时会输出“no”。

**AlwaysYes.exe outputs**

| 输入文件 | 输出 |
|---|---|
| address-list.docx | no |
| mymovie.mpg | no |
| WINWORD.EXE | no |
| ProgramA.exe | yes |
| ProgramB.exe | no |
| NameSize.exe | yes |
| SizeChecker.exe | no |
| Freeze.exe | no |
| AlwaysYes.exe | no |

AlwaysYes.exe对多种输入的输出。只有永远输出“yes”的是—否程序才会让AlwaysYes.exe输出“yes”——在这里就是ProgramA.exe和NameSize.exe。

在表的倒数第二行，你也许注意到出现了一个之前未曾出现的名为Freeze.exe的程序。Freeze.exe是一款程序，是做计算机程序最令人讨厌的事情之一：它会“冻结”（不管输入是什么）。你可能自己经历过，当一个视频游戏或应用程序看起来锁定（或“冻结”）了，拒绝对更多输入做出响应。在此之后，你唯一的选择就是终止程序。如果这还不奏效，你甚至可能需要关闭电源并重启。（有时候在使用笔记本电脑时，断电要移除电池！）计算机程序冻结的原因多种多样。有时是因为“死锁”，我们在第八章讨论过它。在其他情况中，程序也许忙于执行一项永远不会结束的计算——比如，反复搜索一块从未真正存在的数据。

不管怎样，我们无须理解有关冻结程序的细节。我们只需知道AlwaysYes.exe在给出这么一个程序时该怎么做即可。事实上，AlwaysYes.exe的定义相当仔细，答案也很清楚：AlwaysYes.exe在其输入永远输出“yes”时输出“yes”；反之则输出“no”。因此，当输入是像Freeze.exe一样的程序时，AlwaysYes.exe必须输出“no”，这也是我们在上图倒数第二行看到的结果。

## YesOnSelf.exe：一个更简单的AlwaysYes.exe变体

你也许已经想到，AlwaysYes.exe是一个相当聪明、有用的程序，因为它能分析其他程序并预测它们的输出。必须要承认的是，我并未编写这一程序——我只是描述了在我必须编写它的情况下，它会如何表现。现在我要开始描述另一个名为YesOnSelf.exe的程序。这个程序和AlwaysYes.exe类似，但更简单。与输入文件永远输出“yes”时输

出“yes”不同的是，只有输入文件在运行自身时输出“yes”，YesOnSelf.exe才会输出“yes”；反之则输出“no”。换言之，如果我将SizeChecker.exe作为YesOnSelf.exe的输入，那么YesOnSelf.exe会对SizeChecker.exe进行一些分析，以判定SizeChecker.exe在将自身作为输入运行时会输出什么。正如我们已经知道的（见上一节的表），SizeChecker.exe运行自身的输出为“yes”。因此，YesOnSelf.exe运行SizeChecker.exe也会输出“yes”。你可以使用相同的推理得出YesOnSelf.exe运行其他多种输入的结果。注意，如果输入文件不是一个是—否程序，那么YesOnSelf.exe会自动输出“no”。上表显示了一些YesOnSelf.exe的输出——请尝试验证你理解了表中每一行，因为在继续往下读之前，理解YesOnSelf.exe的行为非常重要。

**YesOnSelf.exe outputs**

| 输入文件 | 输出 |
|---|---|
| address-list.docx | no |
| mymovie.mpg | no |
| WINWORD.EXE | no |
| ProgramA.exe | yes |
| ProgramB.exe | no |
| NameSize.exe | yes |
| SizeChecker.exe | yes |
| Freeze.exe | no |
| AlwaysYes.exe | no |
| YesOnSelf.exe | ??? |

YesOnSelf.exe运行多个输入的输出。唯一生成“yes”的输出就是运行自身时输出“yes”的是—否程序，即ProgramA.exe、NameSize.exe和SizeChecker.exe。表中最后一行有点神秘，因为看起来哪种输出都正确。文字部分细致地讨论了这一情况。

我们要多注意两件和这个相当有趣的程序YesOnSelf.exe有关的事情。第一，看一下上表最后一行。YesOnSelf.exe运行自身的输出应该是什么？幸运的是，只有两种可能性，因此我们逐个思考。如果输出

是“yes”，我们知道（根据YesOnSelf.exe的定义），YesOnSelf.exe在运行自身时应输出“yes”。这有点饶舌，但如果你仔细推理，你会发现所有事情都完美地保持一致，并被引导着得出“yes”是正确答案的结论。

但先别急。假如YesOnSelf.exe运行自身的输出是“no”呢？这意味着（还是根据YesOnSelf.exe的定义）YesOnSelf.exe在运行自身时应该输出“no”。和之前一样，这一声明表现出完美的一致！似乎YesOnSelf.exe能选择性输出。只要其坚持自己的选择，它的答案就会正确。YesOnSelf.exe行为中的神秘自由很快就会成为一个相当巨大的冰山之一角，但目前我们还不准备更深入探讨这一问题。

第二，我并没有编写略显复杂的AlwaysYes.exe这一程序。我所做的只是描述其行为。然而，请注意，如果假设我确实编写了AlwaysYes.exe，那么创建YesOnSelf.exe就会很容易。为什么？因为YesOnSelf.exe要比AlwaysYes.exe更简单：YesOnSelf.exe只需检验一个可能的输入，而非所有可能的输入。

## AntiYesOnSelf.exe：YesOnSelf.exe的反面

现在休息一会儿，记住我们想要到达的目标。本章的目标是证明查找崩溃的程序不可能存在。但我们现在的目标没这么高尚。我们只是尝试发现一个这种程序不可能存在的例子。这将是朝向我们最终目标的一块踏脚石，因为一旦了解如何证明某个特定的程序不可能存在，就可以合理地直接将相同的技巧运用在崩溃寻找程序上。好消息是，我们非常接近这个踏脚石目标。我们将再多研究一个是—否程序就完成任务了。

新程序被称为“AntiYesOnSelf.exe”。正如其名字所暗示的，这个程序和YesOnSelf.exe非常相似——事实上，两个程序一模一样，只是

输出逆反了。如果YesOnSelf.exe对某个特定输入输出“yes”，那么AntiYesOnSelf.exe会对同一输入输出“no”。如果YesOnSelf.exe对某个输入输出“no”，AntiYesOnSelf.exe会对同一输入输出“yes”。

无论输入文件是否是一个是—否程序，AntiYesOnSelf.exe都会回答这一问题：

**输入程序在自身上运行时是否输出“no”？**

对AntiYesOnSelf.exe行为的简单描述。

尽管这对完整精确地定义AntiYesOnSelf.exe行为有利，但更细致地说明其行为仍然助益多多。还记得YesOnSelf.exe对于在自身上运行时输出“yes”的输入输出“yes”，反之输出“no”吧。因此，AntiYesOnSelf.exe对于在自身上运行时输出“yes”的输入输出“no”，反之输出“yes”。AntiYesOnSelf.exe会对其输入问下列问题：“输入文件在自身上运行时是不是不会输出‘yes’？”

不得不承认，对AntiYesOnSelf.exe的描述又是个绕口令。你也许会想，换种说法“输入文件在自身上运行时是否输出‘no’”要简单些。为什么这么问不正确？为什么我们要在“不输出‘yes’”上啰唆，而不用更简单的声明“输出‘no’”？答案是程序有时会做输出“yes”或“no”之外的事情。因此，如果某人告诉我们，某个特定程序不输出“yes”，我们不能自动得出结论它会输出“no”。比如，程序可能输出乱码，甚至冻结。不过，我们可以对一种特殊情况得出一个更强的结论：如果我们预先得知一个程序为是—否程序，那么我们就知道这个程序永远不会冻结或生成乱码——程序永远会以输出“yes”或“no”结束。因此，对于是—否程序而言，在“不输出‘yes’”上啰唆相当于更简单的声明“输出‘no’”。

因此，最终我们能对AntiYesOnSelf.exe的行为给出非常简单的描述。不管输入文件是不是一个是—否程序，AntiYesOnSelf.exe都会回答这个问题：“输入程序在自身上运行时是否输出‘no’？”对AntiYesOnSelf.exe行为的这一阐释在后面非常重要，所以我在上框中列出了它。

考虑到我们在分析YesOnSelf.exe上所做的工作，为AntiYesOnSelf.exe制作一张输出表变得格外容易。事实上，我们可以直接复制上文YesOnSelf.exe outputs的表，将所有“yes”输出改成“no”，反之亦然。这么做得到的结果就是上面这张表。和前面一样，最好逐条审视表中每一行，确认你同意输出栏中每一项。当输入文件为是—否程序时，你可以用前页框中的简单阐释，而非弄懂更早之前给出的更复杂的解释。

**AntiYesOnSelf.exe outputs**

| 输入文件 | 输出 |
| --- | --- |
| address-list.docx | yes |
| mymovie.mpg | yes |
| WINWORD.EXE | yes |
| ProgramA.exe | no |
| ProgramB.exe | yes |
| NameSize.exe | no |
| SizeChecker.exe | no |
| Freeze.exe | yes |
| AlwaysYes.exe | yes |
| AntiYesOnSelf.exe | ??? |

AntiYesOnSelf.exe运行多个输入的输出。根据定义，AntiYesOnSelf.exe会生成YesOnSelf.exe相反的答案，因此这张表——除了最后一行——和YesOnSelf.exe的一样，只是“yes”输出换成了“no”，反之亦然。如文中讨论的一样，判定最后一行极其困难。

你能从表中最后一行看出，当我们尝试计算AntiYesOnSelf.exe在自身上运行时的输出时，一个问题产生了。为帮助我们分析这一问题，让我们进一步简化AntiYesOnSelf.exe在上页框中的描述：与其考

虑将所有可能的是—否程序当作输入，我们将集中精力于AntiYesOnSelf.exe将自身作为输入运行时发生的情况。因此，框中以斜体表示的问题“输入程序会……”可以转述为“AntiYesOnSelf.exe会……”——因为输入程序即AntiYesOnSelf.exe。这是我们需要的最终阐释，因此它也在下面框中展示。

现在我们准备好得出AntiYesOnSelf.exe在自身上运行的输出。输出存在两种可能性（“yes”和“no”），因此完成这一问题应该不会太难。我们将逐一处理每个例子。

无论输入文件是否是一个是—否程序，AntiYesOnSelf.exe都会回答这一问题：

**AntiYesOnSelf.exe在自身上运行时是否输出“no”？**

当AntiYesOnSelf.exe将自身作为输入时对其行为的简单描述。注意，这个框只是上文中框的简化版，专门针对输入文件为AntiYesOnSelf.exe的例子。

例1（输出为“yes”）：如果输出为“yes”，那么对框中粗体问题的答案就是“no”。但根据定义，粗体问题的答案是AntiYesOnSelf.exe的输出（再读一遍框中内容以让自己确信）——因此，输出必为“no”。总之，我们刚刚证明了如果输出为“yes”，那么输出为“no”。不可能！事实上，我们遇到了一处矛盾。（如果你对反证法技巧不熟悉，现在是回顾本章先前对这一主题讨论的好时机。我们会在下面几页反复用到这一技巧。）因为我们得到了一处矛盾，我们对输出是“yes”的假设必然不成立。我们已经证明，AntiYesOnSelf.exe在自身上运行时的输出不可能为“yes”。让我们转向另一种可能性。

例2（输出为“no”）：如果输出为“no”，那么对框中粗体问题的答案就是“yes”。但和例1一样，根据定义，粗体问题的答案是AntiYesOnSelf.exe的输出——因此，输出必须为“yes”。换言之，我们

刚刚证明了如果输出为“no”，那么输出为“yes”。和上例一样，我们得到了一处矛盾，因此我们对输出是“no”的假设必然不成立。我们已经证明，AntiYesOnSelf.exe在自身上运行时的输出不可能为“no”。

现在怎么办？我们已经排除了AntiYesOnSelf.exe在自身上运行时仅有的两种可能输出。这同样是一个矛盾：AntiYesOnSelf.exe被定义为一个是—否问题——一个永远以生成“是”或“否”两个输出之一结束的程序。然而，我们刚刚展示了一个特殊输入，而AntiYesOnSelf.exe用这个输入运行并不生成任一上述输出！这一矛盾暗示我们最初的假设为假：因此，并不可能编写一个行为和AntiYesOnSelf.exe相像的是—否软件。

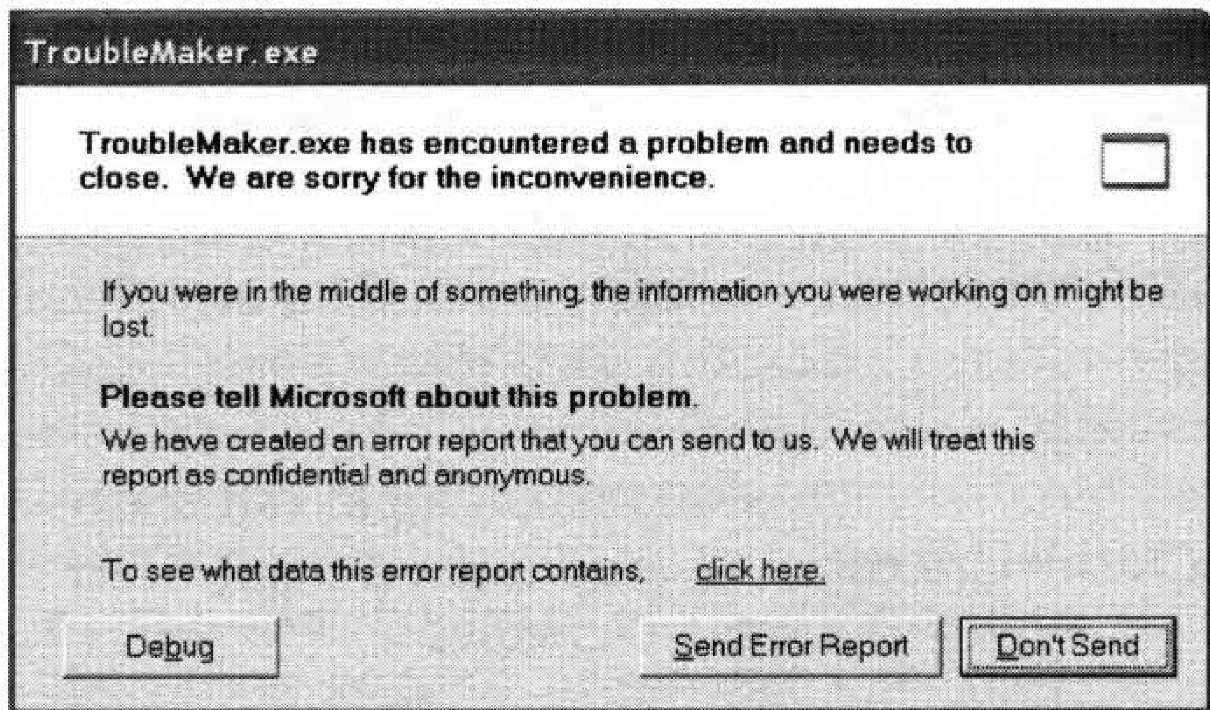
现在你应该知道了，为什么我非常小心地坦然承认自己并未编写过AlwaysYes.exe、YesOn-Self.exe或AntiYesOnSelf.exe中任何一个。我所做的只是描述如果我确实编写了它们，这些程序会如何表现。在上一段中，我们使用了反证法展示AntiYesOnSelf.exe不可能存在。但我们能证明更多：AlwaysYes.exe和YesOnSelf.exe也不可能存在！为什么？你应该能猜到，关键工具还是反证法。回忆我们在上文讨论如果AlwaysYes.exe存在时的情景，对讨论过程稍作修改并得出YesOnSelf.exe是否存在的答案应该会很容易。而如果YesOnSelf.exe存在，那么得出AntiYesOnSelf.exe是否存在的答案也会极其容易，因为我们只须逆转输出即可（将“no”换成“yes”，反之亦然）。总之，如果AlwaysYes.exe存在，那么AntiYesOnSelf.exe也存在。但我们已经知道AntiYesOnSelf.exe不可能存在，因此AlwaysYes.exe也不可能存在。同样的论点也显示YesOnSelf.exe不可能存在。

记住，这整个部分只是一块踏脚石，我们的最终目标是证明崩溃寻找程序不可能存在。本部分更适中的目标是举一些不可能存在的程序的例子。我们已经通过检验三个不同的程序达到了这一目标，那三个程序没有一个可能存在。在这三个程序中，最有趣的要数

AlwaysYes.exe。另两个则相当乏味，这两个例子主要讲以自身为输入运行的程序的行为。另一方面，AlwaysYes.exe是个非常强大的程序，因为只要它存在，就能分析其他任一程序，并告诉我们那个程序是否永远会输出“yes”。但我们现在已经知道，没人能编写出如此聪明、有用的程序。

## 发现崩溃的不可能性

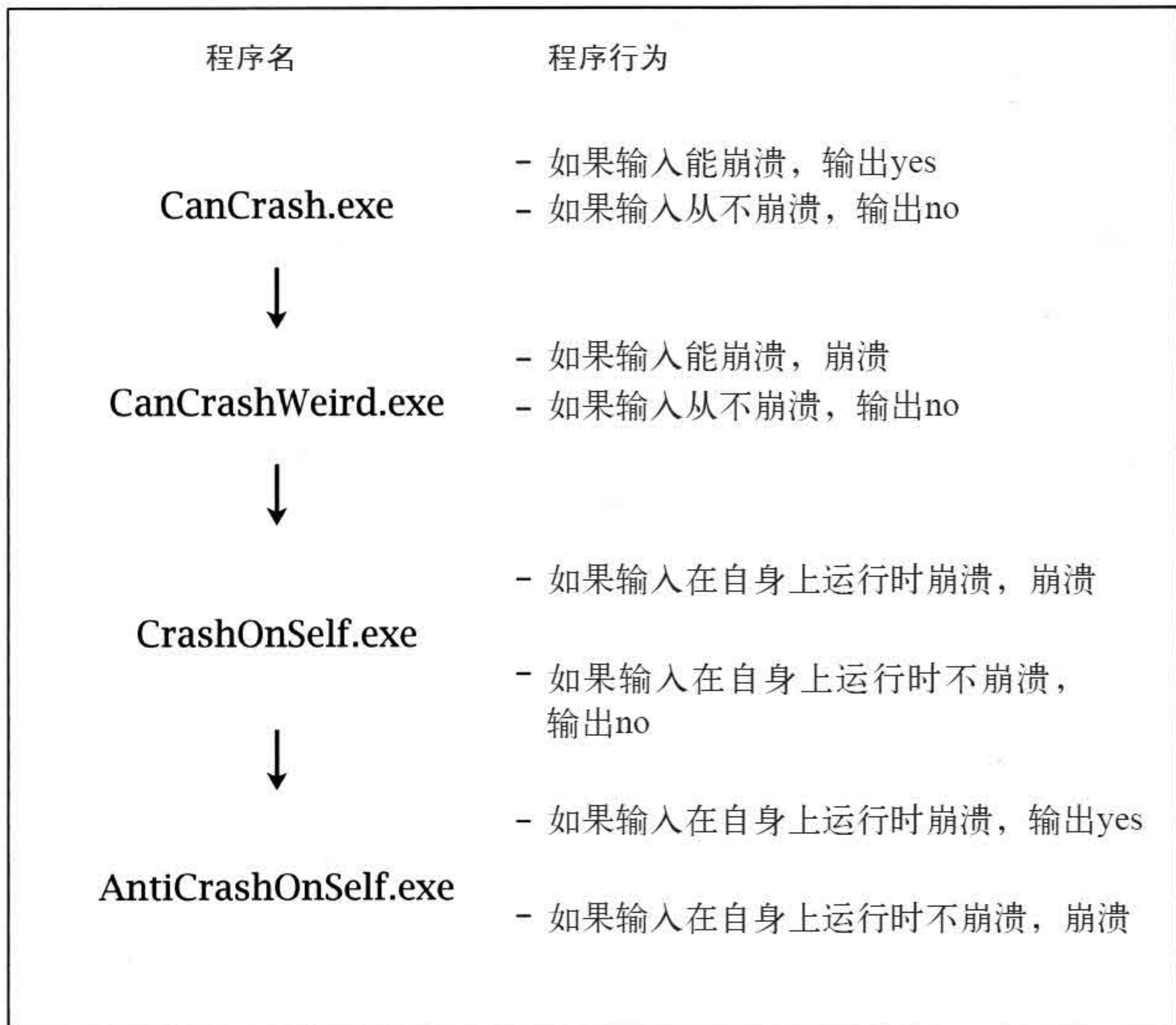
我们终于准备好开始探讨一个程序是否存在的证据，这个程序能成功分析其他程序并判定它们是否会崩溃：具体来说，我们要证明这样一个程序不可能存在。在读完上面几页后，你也许猜到了我们会用反证法。我们一开始会假设存在这样一个程序：有些名为CanCrash.exe的程序能分析其他程序并判定它们是否会崩溃。在对CanCrash.exe做一些奇怪、神秘、令人高兴的事情后，我们会碰到一处矛盾。



某种操作系统崩溃后的结果。不同的操作系统以不同方式处理崩溃，但我们在遇到崩溃时都知道。这个故意编写出来的TroubleMaker.exe会导致崩溃，证明故意导致崩溃很容易实现。

证明过程中有一步要求我们更改一个运行良好的程序，让程序在特定情况下崩溃。我们如何做到这件事？事实上，这件事很容易。程序崩溃的原因很多。其中一种较为常见的原因是程序尝试除以零。在数学中，用任何数除以零得到的结果都是“未定义”（undefined）。在计算机中，“未定义”是个严重错误，程序不能继续就会崩溃。因此，让程序故意崩溃的一种简单方法是，在程序中插入几段会让程序除以零的多余指令。事实上，这也正是我在上图列举TroubleMaker.exe例子的原因。

现在我们开始证明崩溃发现程序不可能性的主要过程。下图总结了论证流。我们一开始假设存在是—否程序CanCrash.exe，如果作为输入的程序会在某种情况下崩溃，CanCrash.exe就会输出“yes”并结束；如果输入程序永不会崩溃，则CanCrash.exe就会输出“no”并结束。




一条证明四个崩溃侦测程序不可能存在的序列。最后一个程序AntiCrashOnSelf.exe 很显然不可能，因为它在自身上运行时会产生矛盾。然而，每个程序都能通过对上一个程序作出小改变轻易得到（如箭头所示）。因此，这四个程序都不可能存在。

现在我们对CanCrash.exe做些诡异的改变：和输出“yes”不同的是，我们会让CanCrash.exe崩溃。（正如上面所讨论的，很容易通过故意除以零做到这一点。）让我们称改变后的程序为CanCrashWeird.exe。因此，如果输入会崩溃，那么CanCrashWeird.exe这个程序也会故意崩溃（形成的对话框外观类似于上图），如果输入永不会崩溃，则CanCrashWeird.exe会输出“no”。

图中显示的下一步是将CanCrashWeird.exe转换成一个更模糊的程序，我们将其称为CrashOnSelf.exe。这个程序和上一部分的YesOnSelf.exe一样，只关注程序在将自身作为输入时运行的表现。特别要说明的是，CrashOnSelf.exe会检测其输入程序，如果输入程序能在自身上运行，则CrashOnSelf.exe会故意崩溃。反之，CrashOnSelf.exe会输出“no”。注意，从CanCrashWeird.exe得到CrashOnSelf.exe很容易：这一过程和我们在第上文讨论的将AlwaysYes.exe转换成YesOnSelf.exe的过程一样。

图中四个程序序列的最后一步就是将CrashOnSelf.exe转换成AntiCrashOnSelf.exe。这简单的一步违背了程序的行为：如果其输入在自身上运行时崩溃，AntiCrashOnSelf.exe输出“yes”。但如果输入在自身上运行时不崩溃，AntiCrashOnSelf.exe就会故意崩溃。

现在我们抵达了生成一处矛盾的地方。AntiCrashOnSelf.exe将自己作为输入运行时会输出什么？根据其自身的描述，如果输入崩溃，AntiCrashOnSelf.exe就会输出“yes”。（矛盾，因为如果AntiCrashOnSelf.exe已经崩溃，它就不能成功输出“yes”并结束。）还是根据其自身的描述，如果输入不崩溃，则AntiCrashOnSelf.exe应崩溃——这一点也自相矛盾。我们已经排除了AntiCrashOnSelf.exe两种可能的行为，这也意味着AntiCrashOnSelf.exe一开始就不可能存在。

最后，我们可以用上页图中显示的转换链证明CanCrash.exe也不可能存在。如果它确实存在，我们就能按图中箭头所示将其转换成AntiCrashOnSelf.exe——但我们已经证明AntiCrashOnSelf.exe不可能存在。这是矛盾之处，因此我们假设CanCrash.exe存在必为假。

## 停机问题和不可判定性

这就对我们经历计算机科学中最成熟、最伟大的问题之一进行了总结。我们已经证明，绝对不可能有人能编写出一个像CanCrash.exe这样的程序：分析其他程序并辨认程序中可能导致其崩溃的所有可能漏洞。

事实上，当理论计算机科学创始人阿兰·图灵首次于20世纪30年代证明一个像这样的结果时，他对漏洞或崩溃完全不关心。毕竟，当时还没有电子计算机。相反，图灵对一个已有计算机程序最终能否生成答案很感兴趣。一个联系紧密的问题是：一个已有计算机程序是否会结束——或者说它是否会永远计算下去，而不生成一个答案？这个已有计算机程序最终是否会结束或“停止”的问题就是著名的停机问题（the halting problem）。图灵的伟大成就在于，计算机科学家称图灵从停机问题得到的变体问题“不可判定”。不可判定问题指不能通过编写计算机程序解决的问题。因此，另一种表述图灵结果的方式是：你不能编写一个名为AlwaysHalts.exe，输入永远停止时输出“yes”，反之输出“no”的计算机程序。

从这个角度看，停机问题和本章处理的问题非常相似，我们可以把本章处理的问题称为崩溃问题。我们证明了崩溃问题的不可判定性，但你基本上可以使用相同技巧证明停机问题也不可判定。而且，正如你所猜测的，计算机科学还有许多其他问题不可判定。

## 不可能程序的应用有哪些？

我收录这一章是故意作为先前章节的对照。前面每一章都捍卫了一个绝佳思想，每个思想都会让计算机更强大，对人类更有用。在本章，我们见识了计算机最基础的限制之一。我们见识到，有些问题根本不可能通过计算机解决，不管计算机有多强大或人类程序员有多聪

明。这些不可判定问题包括潜在的有用任务，如分析其他程序以发现它们是否会崩溃。

这一奇怪的事实——有时甚至先验——有什么影响？不可判定问题的存在会影响我们在实际中使用计算机吗？人类在人脑中进行的计算呢，人脑也不能处理不可判定问题吗？

## 不可判定性和计算机使用

让我们首先来探讨一下不可判定性对计算机使用的实际影响。答案很简短：不，不可判定性对计算的日常实践没有太大影响。原因有两个。第一，不可判定性只关注计算机程序能否生成答案，并不考虑我们需要等答案多久。然而，在实践中，效率问题（也就是你必须等答案多久）极其重要。有许多可判定任务还没有高效算法。其中最著名的要数旅行商问题（Traveling Salesman Problem），简称为TSP。TSP用现代术语表述如下：假设你必须飞往很多城市（假设是20个或30个或100个）。你应该采用哪种顺序访问城市才能让飞行费用最少？我们知道这个问题是可判定的——事实上，只要少量经验的新手程序员就能编写一个计算机程序寻找通过这些城市最便宜的航线。问题是程序可能要花数百万年时间来完成这项工作。在实践中，这并不够好。因此，问题可判定这一事实并不意味着我们可以在实际中解决它。

不可判定性实际效果有限的第二个原因是：我们在大部分时间里都能很好地解决不可判定问题。本章的主要例子就对此进行了很好的展示。我们按照一个详尽的论证过程证明，没有计算机程序能发现所有计算机程序中所有的漏洞。但我们仍然能尝试编写一个漏洞发现程序，希望其发现大多数计算机程序中绝大多数漏洞。事实上，这是计算机科学中一个非常活跃的研究领域。我们在过去几十年所见到的软

件可靠性提升部分得益于崩溃发现程序的进步。因此，通常能为不可判定问题找到非常有用的部分解决方案。

## 不可判定性和人脑

不可判定问题的存在对人类思考过程有影响吗？这一问题会直接引出哲学中一些高深的经典问题，如意识的定义，理智和大脑的区别等。不管怎样，我们对一件事很清楚：如果你相信人脑在原则上能被计算机模拟，那么人脑就会和计算机受相同的限制。换言之，会存在人脑无法解决的问题——不管这个人脑有多聪明或经过多么良好的训练。这一结论紧随本章主要答案之后出现。如果人脑能被计算机程序模拟，而人脑能解决不可判定问题，那么我们也可以用计算机模拟人脑解决不可判定问题——这与计算机程序不能解决不可判定问题的事实相矛盾。

当然，我们是否能让计算机精确模拟人脑的问题还远未解决。从科学观点来看，人脑和计算机之间似乎没有什么基本壁垒，因为化学和电子信号在人脑中传输的低级细节很好理解。另外，多种哲学论据暗示，人脑创造“理智”的物理过程在性质上与计算机能模拟的任何物理系统有所不同。这些哲学论据形式多样，基于我们的自省和直觉能力，或对灵性的追求。

这个问题与阿兰·图灵在1937年发表的有关不可判定性的论文之间有一段迷人的联系。这篇论文被许多人视为让计算机成为一门学科的基石。不幸的是，这篇论文的标题相当模糊：标题以听起来很乏味的短语“关于可计算数字……”（On computable numbers）开头，结尾却以“……及判定问题的应用”（with an application to the Entscheidungsproblem）戛然而止。（我们在这里不会集中讨论标题的第二部分！）在20世纪30年代，“computer”（计算机）这个单词的意义和现在完全不同。对于图灵来说，一台“计算机”就是一个人，用铅笔和纸做些算术。因此，图灵论文标题中的“可计算数字”原则上是能

被人计算的数字。但为辅助他的论据，图灵描述了一种同样能计算的特殊机器（对于图灵而言，一台“机器”就是我们所称的“计算机”）。论文部分内容旨在展示特定计算不可能由这些机器执行——这就是我们已经细致讨论过的不可判定性论证。但同一论文的另一部分提出了一个细致且有说服力的论据，图灵的“机器”（请当作计算机来读）能执行由“计算机”（请当作人来读）执行的任何计算。

你也许开始能理解为何夸大叙述图灵“关于可计算数字……”论文潜在本质很困难了。这篇论文不仅定义及解决了计算机科学中一些最基本的问题，还深入哲学领域，举出了一个具有说服力的例子，表明人类思考过程可以被计算机模拟。（记住，当时计算机还没有被发明！）在现代哲学用语中，认为所有计算机（还有可能包括人）具有相同计算能力的想法以邱奇—图灵论题（Church–Turing thesis）而闻名。这一名称同时向阿兰·图灵和阿隆佐·邱奇致敬。阿隆佐·邱奇（之前提到过）单独发现了不可判定问题的存在。事实上，邱奇比图灵早几个月发表自己的成果，但邱奇的公式更为抽象，且并未详尽提及由机器执行的计算。

对邱奇—图灵论题有效性的争论不断升温。但如果按照最强版本的邱奇—图灵论题所持有的观点，那么并非只有计算机向不可判定性限制俯首称臣。同样的限制不仅适用于我们指尖的精灵，而且也适用于精灵背后的精灵：我们的理智。

# 第十一章　结论——更多在你指尖的精灵

我们只能望见前方不远处，但我们能看到有许多事情需要完成。

——阿兰·图灵，
《计算机器与智能》，1950年

很幸运，我于1991年参加了伟大理论物理学家史蒂芬·霍金（Stephen Hawking）举行的公开演讲。在这场名为“宇宙未来”（The Future of the Universe）的宏大演讲期间，霍金信心满满地预测，宇宙至少还会膨胀100亿年。他还扮着苦相补充道：“当我的观点被证明错误时，我应该不在了。”对我来说，不幸的是，有关计算机科学的预测并没有宇宙学家能得到的100亿年保质期。我做的任何预测都有可能在有生之年被反驳。

但这不应该阻止我们思考计算机伟大思想的未来。我们探索过的伟大算法会永远“伟大”吗？其中一些会过时吗？会有新的伟大算法出现吗？要回答这些问题，我们需要更多地像历史学家而非宇宙学家一样思考。这让我想起了许多年前的另一次经验，在电视中观看由备受争议的牛津知名历史学家A.J.P.泰勒所做的一些演讲。在那个系列演讲末尾，泰勒直接回答了会否有第三次世界大战的问题。他认为答案为“是”，因为人类很有可能“在未来像过去所做过的一样行事”。

因此，让我们跟随A·J·P·泰勒的领导，向历史的宏大鞠躬。本书描述的伟大算法得自贯穿20世纪的事件及发明。似乎假设21世纪以类似

步伐前进很合理，每隔20或30年就有一类新算法崭露头角。在一些情况中，这些算法会具备惊人的原创性，是科学家们梦寐以求的全新技术。公钥加密和相关的数字签名算法就是这类算法的例子。在其他情况中，算法也许已经在实验社区存在了一段时间，等待借助合适的新技术潮流让它们得到广泛应用。索引和排名的搜索算法就属于这类：类似算法在名为信息检索的领域存在了多年，但网络搜索现象让这些算法变得“伟大”——在普通计算机用户日常使用的意义上。当然，这些算法为自己的新应用而进化；PageRank算法就是个好例子。

注意，新技术的出现并不一定会导致新算法产生。想想笔记本电脑在20世纪80年代及90年代的显著增长。通过极大增加可访问性和便携性，笔记本电脑革命化了人们使用计算机的方式。笔记本电脑还引发了多个领域极其巨大的进步，如屏幕技术和电池管理技术。但照我看，自笔记本电脑革命以来再没有伟大算法出现。相反，互联网的出现就是一项导致许多伟大算法出现的技术：互联网通过提供搜索引擎能存在的基础架构，让索引和排名算法得以跻身伟大算法行列。

因此，尽管技术革新不可置疑的加速继续在我们周围出现，但并不能保证新的伟大算法的出现。事实上，相反方向上有一种强大的历史力量在作用，暗示算法创新的步伐将在未来减慢。我说的是计算机科学作为一门科学学科开始成熟。同物理学、数学和化学等领域相比，计算机科学非常年轻：它于20世纪30年代发端。因此，20世纪发现的伟大算法也许已经是唾手可得的硕果，在未来发现广泛应用的精巧算法将变得越来越困难。

因此，我们有两个相互竞争的效果：新技术提供的新活动（niche）时常为新算法提供空间，而该领域的逐渐成熟会缩小这些机会。总之，我倾向于认为这两种效果会彼此中和，让未来新的伟大算法缓慢但稳定地出现。

## 一些可能的伟大算法

当然，一些新的伟大算法会以完全出人意料的方式出现，现在也不可能对未来的伟大算法说些什么。但一些现存的活动和技术有很清晰的潜力。其中一个明显趋势是人工智能在日常生活中的逐渐使用（特别是图形识别），如果有任何令人震惊的崭新算法瑰宝在这一领域出现，会让人非常着迷。

另一个潜力非凡的领域是一类名为“零知识协议”（zero knowledge protocols）的算法。这些协议使用一种特殊的加密方法，以实现一些比数字签名更令人惊讶的事情：它们能让两个或更多实体将信息组织起来，而不用显示任何单块信息。零知识协议的潜在用途之一是在线拍卖。通过使用该协议，竞拍者能以加密方式向彼此提交各自的竞价，从而判定赢得竞拍的人，但任何其他竞价的信息都不会向任何人展示！如果零知识在现实中运用的话，这样一个聪明的想法肯定会轻易满足我的伟大算法行列标准。但目前为止，它们还未被广泛使用。

另一种获得众多学术研究但实际应用有限的思想是一种名为“分布式哈希表”（distributed hash table）的技术。这些表是一种在点对点系统中——一个没有中央服务器引导信息流的系统——存储信息的精巧方法。然而，在写作本书时，许多号称点对点的系统实际上仍在一些功能上使用中央处理器，因此无需依赖分布式哈希表。

“拜占庭容错”（Byzantine fault tolerance）技术也属于这一类：这是一种令人惊讶但美丽的算法，不过仍然不能算作伟大，因为没什么人采用。拜占庭容错允许特定计算机系统耐受任何种类的错误（只要同时不出现太多错误）。这与平常的容错概念——系统能幸免的错误更轻微，如磁盘驱动器永久失效或操作系统崩溃——相反。

## 伟大算法会消逝吗?

除了推测什么算法会在未来步入伟大行列之外，我们也许还会想，目前的“伟大”算法中——我们想都不想经常用到的不可取代的工具——是否有一些会逐渐丧失其重要性。历史在这里也能指引我们。如果我们将注意力限制在特定算法上，算法肯定会丧失重要性。最明显的例子在密码学中；在发明新加密算法的研究人员和发明破解这些算法安全性的研究人员之间，进行着一场不间断的军备竞赛。作为一个特例，思考一下所谓的加密哈希函数。名为MD5的哈希函数是一个官方互联网标准，自20世纪90年代开始广泛使用，然而自此以后MD5的重大安全缺陷被不断发现，人们不再推荐使用MD5。类似的，我们在第九章讨论过这一事实：如果搭建一个合理体积的量子计算机成为可能，RSA数字签名机制将轻易被攻破。

然而，我认为用这样的例子来回答我们的问题太片面。的确，MD5有缺陷（顺便说一下，其主要继承者SHA-1也是），但这并不意味着加密哈希函数的中心思想就无关紧要了。的确，这类哈希函数运用得非常广泛，也还有许多未被破解的算法。因此，我们用足够宽广的视角看待这一情况，并准备好在获取算法主要思想时适应算法特例，现今许多伟大算法似乎不大可能在未来丧失其重要性。

## 我们学到了什么?

能从本书展示的伟大算法中得出什么通用主题吗？其中一个主题是，所有大思想都能在不需要任何计算机编程或其他计算机科学知识的情况下得到解释。作为本书作者，我感到很惊讶。当我开始着手这本书时，我以为能将这些伟大算法分成两类。第一类是一些核心思想中运用了简单但聪明的把戏的算法——这些把戏无须任何技术性知识

就能得到解释。第二类算法与高等计算机科学思想联系紧密，不能向没有该领域背景知识的读者解释。我原计划通过讲些第二类算法（可能）有趣的历史轶事来涵盖一些这类算法，解释它们的重要应用，并言之凿凿地称这些算法非常精巧，即便我不能解释它们如何运行。当我发现自己选择的所有算法都归属为第一类时，想想我有多高兴、多惊讶！当然，我省略了许多重要的技术细节，但在每个例子中，让整件事奏效的关键机制可以通过使用非专业性概念解释。

本书所有算法的另外一个重要的通用主题是，计算机科学领域并不仅仅是编程。每当我教授一门入门计算机科学课程，我都会要求学生告诉我，他们认为计算机科学究竟是什么。到目前为止，最常见的回答是“编程”，或类似的“软件工程”。当我继续要求他们说出计算机科学的其他方面时，许多学生被难住。但接下来回答的学生通常会说些和硬件有关的东西，如“硬件设计”。这是对计算机科学家真正从事的工作的一种流行误解的强力证据。读过这本书后，我希望你能对计算机科学家思考的问题，他们提出的解决方案有更具体的了解。

在这里举个简单类比会很有帮助。假设你遇到了一位主要研究兴趣是日本文学的教授。极有可能这位教授能说、读以及写日文。但如果你被要求猜测这位教授在进行研究时在什么上花了最多时间进行思考，你不会猜是“日本语言”。尽管，日本语言是研究组成日本文学的主题、文化和历史的必要知识。另一方面，说一口完美日语的人也许完全无视日本文学（日本可能有数百万这样的人）。

计算机编程语言和计算机科学主要思想之间的关系与上面的情况很相似。要应用并实验算法，计算机科学研究者需要将算法转换成计算机程序，而每个程序都由Java、C++或Python等编程语言编写。因此，编程知识是计算机科学家所必需的，但却只是前提：在见识了本书中的伟大算法以后，我希望读者们能对这一区别有更清楚的理解。

# 旅程终点

我们已经抵达深奥但也日常的计算世界的终点。我们实现自己的目标了吗？你和计算设备的互动会因此不同吗？

下次当你访问一个安全网站时，你也许会想知道谁为其可信度担保，并检查你的网络浏览器侦测到的数字证书链（第九章）。或者当你下次遇到在线交易因为未知原因失败时，你会深深感激，而不是困惑，因为你知道数据库一致性会确保你不会为订购失败的东西付费（第八章）。或者未来某天你会自娱自乐地说："如果计算机能为我做这件事该多好啊"——但你知道这不可能，因为使用在崩溃发现程序中用到的相同方法，你想让计算机做的任务被证明不可判定（第十章）。

我肯定你能想到更多例子，在这些例子中，伟大算法的知识也许能改变你与计算机互动的方式。然而，正如我在前言中仔细说明的，这并非本书的主要目的。我的主要目的是让读者有足够多和伟大算法有关的知识，让他们在一些日常计算任务时能遐想片刻——就像一名业余天文学家高度崇敬夜空一样。

只有你——我的读者——知道我是否成功地实现了这一目标。但有件事很肯定：你自己的个人精灵就在你的指尖。请自由使用。

# 致谢

你这条我走着的，并还在四面环顾着的路啊，我相信你不知眼下这一点，我相信这里也还有许多看不见的东西。

——沃尔特·惠特曼，

《大路歌》（Song of the Open Road）(注)

许多朋友、同事和家人阅读了部分或所有草稿。他们是亚历克斯·贝茨、威尔森·贝尔、麦克·巴罗斯、沃尔特·克劳米亚克、迈克尔·艾萨德、阿拉斯泰尔·麦考密克、瑞雯·麦考密克、妮可莱塔·马里尼–迈欧、弗兰克·麦克谢里、克莉斯汀·米切尔、伊莉亚·米罗诺夫、温迪·波拉克、朱迪斯·波特、科顿·西勒、海伦·塔卡克斯、库纳尔·塔尔瓦、蒂姆·瓦尔斯、乔纳森·瓦勒、乌迪·维德和奥利·威廉姆斯。这些读者的建议极大地提升了书稿的质量。两位匿名审阅者的评论也让书稿得到了很大提升。

克里斯·毕晓普给予了我鼓励和建议。汤姆·米切尔给予了我在第六章使用他的图片和源代码的授权。

在孵化本项目并让它开花结果上，本书编辑维琪·克恩和她在普林斯顿大学出版社的同事居功至伟。

我在迪金森学院数学与计算机科学部门的同事对本书予以了持久支持和帮助。

迈克尔·艾萨德和麦克·巴罗斯向我展示了计算的愉悦和美丽。安德鲁·布莱克教会了我如何成为一名更好的科学家。

我的妻子克莉斯汀一直在支持我。

我向以上所有人致以最深的谢意。本书并我的爱献给克莉斯汀。

---

1. 选自赵萝蕤译本。